UNIVERSITÉ DE NICE / SOPHIA-ANTIPOLIS ,
UFR Sciences

École Doctorale STIC
Sciences et Technologies de l'Information et de la Communication

Thèse de Doctorat

Discipline : INFORMATIQUE

Présentée et soutenue publiquement le 14 Décembre 2007, par :

Julien BOUCARON

Titre

Modélisation formelle de systèmes Insensibles à la Latence et ordonnancement.

JURY :

M. Michel Auguin	Directeur de Recherche CNRS	**Président**
M. Alain Girault	Directeur de Recherche Inria	**Rapporteur**
Mme. Alix Munier Kordon	Professeur à Univ. Paris 12	**Rapporteur**
M. Marc Benveniste	ST Microelectronics à Rousset	**Examinateur**
M. Robert de Simone	Directeur de Recherche Inria	**Directeur de Thèse**

À mon grand-père Louis BOUCARON pour m'avoir
donné le goût de faire des choses,
À ma tendre et douce Katia pour ton soutien indéflectible et ton amour,
À ma famille et à mes amis pour tout le reste

Remerciements

Je tiens à remercier ici :

– Robert de Simone, pour m'avoir permis d'effectuer cette thèse sous sa direction.
– Jean-Vivien Millo, Benoît Ferrero pour les discussions nombreuses et animées sur les histoires tordues d'ordonnancement et de liveness ;-).
– Marc Benveniste et ST Microelectronics à Rousset, pour avoir créé le projet ForComment qui a permis de financer ce travail.
– Michel Auguin pour avoir accepté d'être président de mon jury.
– Alix Munier Kordon et Alain Girault pour s'être intéressés à mon travail et pour avoir accepté d'être rapporteurs pour ma thèse.
– Charles André, Daniel Gaffé, Dumitru Potop-Butucaru et Frédéric Boussinot pour porter la divine parole du synchrone.
– Je tiens également à remercier chaleureusement Anthony Coadou, Fabrice Peix, Olivier Tardieu, Éric Vecchie, Arnaud Cuccuru, Frédéric Mallet, Marie Agnès Péraldi et toute l'équipe Aoste de Rocquencourt à Sophia.

Enfin, merci à toutes les personnes de l'INRIA, l'I3S qui de près ou de loin ont été en relation avec moi et ont permis d'une façon ou d'une autre que j'achève ce manuscrit.

Résumé

Mots clefs :

Synchrone, SDF, Marked Event Graphs, Ordonnancement, Latency Insensitive

Cette thèse présente de nouveaux résultats liant la théorie des systèmes dits insensibles à la latence, à une sous-classe des réseaux de Pétri dénommée Marked Event Graph et son extension dite Synchronous Data Flow. Ces travaux sont intimement associés avec le problème d'ordonnancement général dénommé problème central répétitif.

Nous introduisons les modèles synchrones, Marked Event Graphs, Synchronous Data Flow (SDF) et Latency Insensitive.

Après, nous discutons des liens existants entre les modèles synchrones, Marked Event Graphs et Latency Insensitive ; nous montrons que le modèle Latency Insensitive est un cas particulier du modèle Marked Event Graph.

Nous présentons ensuite une implémentation vérifiée formellement de Latency Insensitive.

Après, nous rappelons un résultat connu : tout Marked Event Graph ayant au moins une partie fortement connexe (et s'évaluant avec une règle d'exécution *As Soon As Possible* (ASAP)) a un comportement ultimement répétitif : c'est à dire qu'il existe un ordonnancement statique. À partir de ce résultat, nous construisons une technique d'ordonnancement particulière dénommée *Égalisation* qui altère virtuellement la topologie des communications du système afin de ralentir des chemins trop rapides en rajoutant des "registres", tout en conservant les performances en terme de débit du système originel.

Enfin, nous introduisons une notion de contrôle limité au modèle Latency Insensitive, avec des nœuds appelés *select* et *merge* dont les conditions sont connues

et indépendantes des flots de données, plus exactement les conditions d'aiguillage des données sont dirigées par des mots binaires ultimement périodiques (comme dans le cadre de l'ordonnancement statique). Nous effectuons ensuite une abstraction sur le modèle SDF afin de déterminer si le modèle accepte un ordonnancement où la taille de toute place est bornée. Nous pouvons vérifier ensuite la vivacité du système grâce à une simulation, si le modèle originel disposait d'au moins d'une partie fortement connexe.

Finalement, nous concluons et discutons des possibilités de travaux futurs.

Abstract

TITLE : Formal modelling of Latency Insensitive systems and scheduling

Keywords :

Synchronous, SDF, Marked Event Graphs, Scheduling, Latency Insensitive

This PhD thesis introduces new results linking the theory of Latency Insensitive, to a well-known sub-class of Petri Nets called Marked Event Graphs and its extension called Synchronous Data Flow. This work is tightly linked with a well-known problem, called Central Repetitive Problem (workshop scheduling...).

We introduce the Synchronous Models, Marked Event Graphs, Synchronous Data Flow (SDF) and Latency Insensitive.

After, we discuss existing links between the Synchronous Models, Marked Event Graphs, Synchronous Data Flow (SDF) and Latency Insensitive ; we show that the Latency Insensitive model is a special case of the Marked Event Graph model.

After, we recall a well-known result : any Marked Event Graph with at least a strongly connected component (and evaluating with the firing rule *As Soon As Possible (ASAP)*) enjoys an ultimately repetitive behaviour : that is to say that it exists a static schedule. Starting from this result, we build a specific scheduling scheme called *Equalization* that is altering virtually the communication topology in order to slow-down too fast pathes adding some “registers”, while preserving the global performance in throughput of the original system.

Finally, we introduce some limited control in the Latency Insensitive Model, with nodes called *select* and *merge* where conditions are known and indepen-

dant of data flows, more precisely datas are directed by ultimately periodic binary words (just like in the case of the static scheduling). We are creating then an abstraction over the SDF model in order to determinate if the instance of the model accepts a schedule where the size of each place in bounded. We can verify then the liveness of the system through simulation if the original system was having at least a strongly connected component.

Finally, we conclude and discuss possibilities for future works.

Table des matières

Chapitre 1

Introduction

Ce document traite de la conception de systèmes (électroniques) insensibles aux latences (des communications). Plus précisemment on s'attache à pouvoir resynchroniser des spécifications initiales, qui peuvent être totalement synchrones ou totalement asynchrones, tout en respectant des latences arbitrairement données entre les blocs de calculs élémentaires, et ce sans modifier de manière notable la conception de ces blocs eux-mêmes : ce qui permet de les réutiliser et de définir les latences de manière extérieure a posteriori, réglant de ce fait la question dite de "timing closure" des systèmes électroniques.

Nous venons de dire que les spécifications initiales considérées sont indifféremment synchrones ou asynchrones. De manière importante ces spécifications seront avant tout à base de modèles sans conflits ("conflict-free"), dont les comportements sont confluents. Les modèles "de base" mono-horloge de la programmation réactive synchrone, et les graphes d'évènements (Marked Event Graphs [34]) seront donc nos objets de départ privilégiés. La première partie du document reviendra sur ces modèles, leurs propriétés et leurs restrictions. On montre également comment certaines extensions des graphes d'évènements considérant des poids et indications temporelles de latences, peuvent se ramener par expansion aux graphes classiques.

La théorie du "Latency Insensitive Design" (LID), dûe originellement à Luca Carloni [21] et développée par la suite par plusieurs auteurs, consiste à représenter les communications globales, sujettes à latence, par des chaînes de stations-relais qui les divisent en tronçons de latence unitaire. Chaque station-relais doit implanter un protocole de contrôle de flux pour éviter les phénomènes de congestion quand des connexions "rapides" voient leurs données retardées par des connexions plus lentes. En particulier chaque station doit posséder (au moins) deux registres

de stockage pour les données, afin de pouvoir accueillir une valeur de l'amont au moment même où le trafic ne permet pas de propager la valeur précédente en aval de la chaîne de communication. Enfin des conteneurs ("Shell wrappers") doivent venir encapsuler les blocs de calcul élémentaires afin de les piloter pour les exécuter exactement quand toutes les valeurs d'entrée sont parvenues à destination sur toutes les connexions (d'entrée).

Dans une seconde partie du document nous proposons une modélisation formelle par des descriptions réactives synchrones des stations-relais et des shell wrappers. Nous montrons la correction de ce schéma de traduction en définissant par ailleurs que les cellules des stations-relais doivent modéliser des places de capacité bornée (à 2) de graphes d'évènements, et que la sémantique se définit alors par une règle de franchissement "ASAP" (As Soon As Posssible). Nous donnons des propriétés et preuves de correction, tant au niveau du comportement de combinaisons de stations-relais et de conteneurs (qui peuvent être construites et validées à l'intérieur d'environnement de programmation réactif synchrone), que de l'équivalence sémantique entre modèles. Cette dernière partie repose essentiellement sur les résultats d'équivalence "élastique" entre les comportements de modèles synchrones et asynchrones, mais "confluents" (car sans conflits).

Il est par ailleurs bien connu (mais dans d'autres communautés scientifiques) que sous certaines hypothèses naturelles les comportements de graphes d'évènements conduisent sous une politique "ASAP" à des régimes ultimement (k-)périodiques. Ces conditions sont essentiellement l'existence de boucles fermées et le fait que les entrées/sorties du systèmes acceptent les rythmes établis de manière interne. Les régimes k-périodiques peuvent être précisément calculés, ce qui en retour permet dans notre cas de calculer plus spécifiquement les positions où des éléments de stockage et de "bufferisation" sont nécessaires. À ce sujet la modélisation LID et l'introduction des stations-relais améliorent déjà la situation par rapport aux graphes d'évènements simples, en répartissant ces cellules afin que des accumulations importantes ne puissent se produire au cours du temps à des endroits fort différents du système, ce qui pourrait conduire à prévoir la possibilité de les contenir partout. Le calcul des ordonnancements statiques, en établissant les lieux et instants exacts des besoins en stockage temporaire des données (pour un instant individuel), permet de raffiner encore cette allocation. Les résultats de k-périodicité sont dûs essentiellement aux travaux de Carlier et Chrétienne [19], de Cohen, Baccelli, Quadrat *et al.* [8], et de plusieurs auteurs étendant par la suite leurs travaux.

Dans la troisième partie nous étudions l'optimisation de cette allocation de ressources en cherchant à construire des ordonnancements statiques de type k-

périodique qui possèdent des propriétés supplémentaires. Par ce biais on peut espérer minimiser le nombre de registres nécessaires sur les communications à latence, mais aussi à se défaire du protocole de contrôle de flux nécessaire dans la version ordonnancée dynamiquement, et qui induit toute une circuiterie de signalisation. L'objectif est alors d'égaliser au mieux possible les latences entre les blocs de calcul élémentaires, afin de contrôler les différences d'apparitions entre données. Dans un premier temps on cherche à saturer au maximum le système par des latences entières additionnelles (implantées comme registres de type "latch" en électronique) sur les chemins de longueur non critique. Ces latences peuvent être considérées virtuelles, au sens où l'on peut les utiliser (voire les supprimer) pour re-concevoir certains blocs de calcul en relaxant leurs contraintes temporelles. On peut aussi les déplacer en vue d'harmoniser les instants de calcul des divers blocs (par exemple éviter que des instants "forts" de plein calcul alternent avec des moments "faibles" de simple communication au travers du réseau. Ces opérations sont dénommées "recycling" par Carloni et Sangiovanni-Vincentelli [22, 23]. Après ajout de latences entières il reste quand même nécessaire de traiter le cas de latences fractionnelles résiduelles pour obtenir une pleine égalisation. Il s'agit ici du cas où les latences entre chemins différents ne s'accordent qu'en retenant certaines des données (mais pas toutes) au long d'une période du régime stationnaire global du système correspondant à l'ordonnancement statique. Nous proposons ici des constructions, et il reste encore certains problèmes ouverts au sujet de l'optimalité de cette allocation permettant de se passer complètement de la signalisation du contrôle de flux.

Comme il a déjà été indiqué notre approche utilise au départ des spécifications synchrones ou asynchrones, mais surtout confluentes et libres de conflits. Les descriptions de base réactives synchrones ou les graphes d'évènements implantent ceci en interdisant simplement tout choix ou contrôle alternatif entre comportements. Il existe néanmoins des moyens moins radicaux d'assurer la confluence comportementale, comme dans les réseaux de Kahn [47] où le seul non-déterminisme local admis est un non-déterminisme interne (des choix entre branches comportementales qui ne peuvent porter sur l'existence ou la non-existence de données dans les canaux d'entrée, comme c'est le cas par exemple dans les "input guard conditions" d'un langage comme CSP [46]). Le problème principal de réseaux de Kahn est qu'en général l'aspect borné et dimensionnable des besoins en stockage temporaires des connexions de communication y est indécidable.

Dans la quatrième partie de cette thèse nous proposons une spécialisation de réseaux de Kahn abstraits, dans lesquels les composants élémentaires séquentiels sont à état fini, les données abstraites, et les conditions internes de branchement respectent au cours du temps des schémas d'alternance k-périodiques (suivant

la syntaxe même qui justement avait permis de caractériser et manipuler les ordonnancements statiques individuels dans la section précédente). Pour ce faire nous introduisons deux types simples de blocs de base, dénommés respectivement *merge* et *select*, aux fonctions assez largement inverses et permettant de fusionner ou au contraire de démultiplexer des flots de données. Ces blocs permettent d'encoder notre classe simple de réseaux de Kahn (selon une méthode ressemblant par exemple à la traduction de programmes Esterel en circuits synchrones, où le contrôle séquentiel est encodé par des connexions supplémentaires de signalisation). Par ailleurs ces blocs peuvent également permettre de modéliser des nœuds d'interconnexion (switches) au sein du réseau, et peuvent être utilisés en pleine généralité en combinaison avec les blocs de calcul et les connexions à latences des modèles précédents. Les blocs merge et select ont une sémantique formelle définie par leurs transformations sur les mots k-périodiques représentant les ordonnancements statiques et définis dans les sections précédentes. Nous introduisons deux opérateurs sur ces mots, ce qui nous permet de formuler des identités algébriques pour la permutation et la commutation de nœuds merge et select entre eux. Par abstraction de ces noeuds vers un modèle de type Synchronous Data Flow (SDF) [53] , une autre extension des graphes d'évènements également présentée en section 1, nous sommes capables d'établir les conditions exactes sur les relations entre "mots k-périodiques de branchement" gouvernant les choix non conflictuels, afin que les taux de production/consommation de données dans les canaux de connexions s'accordent. La question qui demeure alors est de savoir comment ordonnancer, ou rendre compte d'ordonnancements, à l'intérieur de la k-période qui permet de revenir à la stabilité initiale des positions de données. En utilisant nos lois algébriques précédentes on peut montrer qu'il existe des formes normales de topologies de connexions, utilisant nos nœuds merge et select, et tels que ces problèmes d'ordonnancement à l'intérieur d'une période ne se posent que sur une seule forme simple de réseau (un canal démultiplexé puis remultiplexé). Néanmoins ces formes normales de réseaux sont issues d'expressions "développées" (plutôt que "factorisées"), et ne partagent pas les canaux entre blocs de calcul. Le principal intérêt de notre formalisation est au contraire, à nos yeux, de permettre de poser la question d'un partage optimisé des ressources de connexion sous l'hypothèse (favorable mais très limitative) que les latences de communication et les "patterns" de choix de branchement soient entièrement prédits et connus.

Il reste encore bien entendu de nombreux problèmes ouverts, comme la combinaison efficace des ordonnancements k-périodiques dûs aux latences temporelles avec les branchements k-périodiques abstrayant des choix (non conflictuels) de branchement. L'optimalité du placement des registres fractionnaires à la section 3, et les propriétés qu'elle induirait sur la signalisation de contrôle de flux en sont

un autre exemple. À tous les niveaux l'étude de la complexité algorithmique et des représentations internes efficaces est également un sujet qui, s'il est largement abordé dans la troisième partie du document, devrait être encore largement étendu sur de nombreux points (dans la quatrième partie notamment).

Nos travaux sur la modélisation formelle des stations-relais et conteneurs, ainsi que sur l'égalisation des latences dans le cadre de l'ordonnancement statique, ont fait l'objet d'implantations prototypes menant à un outil interne dénommé K-PASSA (K-Periodic As-Soon as Possible Scheduling and Analysis). Certains prolongements de ces travaux ont été réalisés en commun et de concert avec Jean-Vivien Millo, dont la thèse viendra je l'espère compléter certains éléments sur l'allocation précise de ressources et de cycles d'exécutions dans le cadre de l'ordonnancement statique de systèmes LID.

Chapitre 2

Modèles sous-jacents

Il n'y a pas de simplicité véritable. Il n'y a que des simplifications.
- Léon-Paul Fargue

Nous décrivons le modèle synchrone, quelques langages synchrones et le calcul d'horloge qui est une forme de typage. Nous introduisons le modèle asynchrone Marked Event Graphs, qui est un modèle déterministe et confluent. Puis, nous exposons deux extensions de cette sous-classe particulière des réseaux de Petri, nommées Timed Marked Event Graphs et Synchronous Data Flow (SDF).

2.1 Langages et formalismes synchrones

Les langages synchrones ont émergé au début des années 80 principalement dans des groupes de recherche français pour spécifier, développer formellement des systèmes réactifs et temps réel.

Différents langages ont été créés afin de répondre à des besoins différents à travers des langages textuels tels que Lustre [42], Signal [9, 1], Esterel [17, 10], Lucid Synchrone [26, 25] , ou graphiques comme StateCharts [43], Argos [57, 56], SyncCharts [4, 5] ou même encore des méthodes tabulaires comme SCR [44, 45]. Nous pouvons noter également que la plupart des langages de description de matériel tels que VHDL ou Verilog définissent un sous-ensemble qualifié de *synthétisable* de nature synchrone.

L'objectif initial de ces langages était la description de spécification et la programmation *sûre* de systèmes temps-réel *critiques*, comme nous en trouvons dans les systèmes embarqués (avionique, automobile, robotique, ...etc...) : avion (SCR,

StateCharts, Esterel, SyncCharts), contrôle de centrale nucléaire (Lustre), spécification de matériel (VHDL, Verilog - *NB :* ces 2 langages permettent aussi de simuler et synthétiser du matériel).

2.1.1 Modèle Réactif Synchrone

Les systèmes réactifs sont en interaction permanente avec leur environnement. Lorsque cet environnement modifie un ensemble des entrées du système réactif, ce dernier réagit et modifie à son tour un ensemble de cet environnement en activant ou désactivant des sorties.

Dans les langages synchrones il n'y a pas de notion de temps physique au niveau du calcul et des communications, à chaque *instant* logique le système lit toutes ses entrées et génère toutes ses sorties avant la fin de l'instant. Ainsi nous disons par abus de langage que les sorties sont "synchrones" aux entrées.

Les langages synchrones disposent d'une sémantique formelle et s'appuient sur des modèles mathématiques relativement simples. L'apport principal des langages synchrones est d'introduire du parallélisme *déterministe* grâce à la notion *d'instant*, contrairement au modèle asynchrone. Un autre apport est la possibilité d'exploiter les modèles sous-jacents en vue de preuves, d'analyses et optimisations formelles de programmes. D'un point de vue pratique, les hypothèses synchrones permettent de se concentrer exclusivement sur la correction du comportement, de la fonctionnalité du programme ; il y a abstraction des critères d'implémentation tels que la vitesse, la taille du programme ...etc...

Les langages synchrones peuvent être décomposés en trois catégories : flot de données (dataflow) tel que Lustre [42], contrôle de flot c'est à dire impératif comme Esterel, SyncCharts ; ou encore "relationnel" tel que Signal.

Néanmoins cette catégorisation n'est pas fondamentalement très utile pour comprendre le principe de base du synchrone car finalement ces descriptions peuvent s'unifier vers un modèle unique que nous pouvons qualifier d'*assembleur*. Cet assembleur est une suite d'opérations avec affectations uniques (Single Static Assignement (SSA)) acycliques qui sont partiellement triées formant un ordre partiel. A chaque instant, nous lisons toutes les entrées, nous évaluons toutes les équations de ce modèle dans un ordre total compatible avec l'ordre partiel imposé et nous mettons à jour les sorties et variables d'états (registres).

Nous allons introduire ce modèle plus formellement.

Définition 1 (Modèle Synchrone)**.** Un modèle synchrone est un n-uplet :
$< Clock, Inputs, Outputs, Locals, Regs, Eqns >$, où

- *Clock* est l'horloge échantillonnant les signaux d'entrées et générant les signaux de sorties.
- *Inputs* sont les signaux d'entrées.
- *Outputs* sont les signaux de sorties définis par une et une seule équation.
- *Locals* les signaux intermédiaires entre les *Inputs* et *Outputs* qui peuvent être typés optionnellement (*Type*). Chaque signal local est défini par une et une seule équation.
- *Regs* les signaux émanant/finissant dans un élément de mémorisation qui seront définis dans l'instant courant pour être utilisés lors du prochain instant : *Regs* est une liste de n-uplets $< Current, Next, [Type] >$, où *Current* est la valeur lue (une entrée) sur l'élément de mémorisation qui a été calculée lors de l'instant précédent via le signal *Next* associé.
  NB : Next $\in \{Locals \bigcup Outputs\}$. *Type* est une condition optionnelle, lorsqu'elle est absente de la spécification alors à chaque instant (quand *Clock* présent) *Next* sera évalué. Par contre si la condition de typage est présente dans la spécification alors il faut que tous les signaux présents dans *Next* satisfassent la condition de typage afin que *Next* puisse être évalué.
- *Eqns* l'ensemble des équations reliant les entrées, locaux et sorties. Une équation est définie de la manière suivante :
  $X_i = E(Y_0, ..., Y_n)$ où $X_i \in Outputs \cup Locals \cup Regs.Next$ est la définition du signal qui est soit une sortie, un local ou la valeur au prochain instant d'un registre et $\forall_{j=0}^{j=n} Y_j \in Inputs \cup Locals \cup Regs.Current$ sont des utilisations des entrées, locaux et les valeurs à l'instant précédent de registres.
  L'équation est décrite à l'aide d'opérateurs binaires et unaires : arithmétique, logique. L'équation forme un arbre dont la racine est l'opérateur de définition.

Tous les signaux sont une paire *valeur,présence* à chaque instant où *valeur* est la valeur du message porté par le signal et *présence* indique si le signal est présent $\top$ où absent $\bot$ lors de l'instant courant. On peut souvent abstraire les valeurs et seulement conserver la présence $\top$ et l'absence $\bot$ du signal.

L'horloge *Clock* est distribuée par un réseau particulier nommé *arbre d'horloge*. C'est ce réseau qui assure la synchronisation globale du circuit en générant un temps de référence et le distribuant à chaque registre (le lecteur est invité à lire le chapitre suivant [49] pour des explications détaillées sur le sujet).

À cause des hypothèses sur le temps de calcul et de communication nuls, il faut disposer d'un critère de convergence car malheureusement il peut arriver que nous décrivions une "boucle" qui est potentiellement "infinie" dans le temps. Une boucle est l'existence d'une chaîne d'utilisations de signaux amenant à une défi-

nition d'un signal présent dans cette dîte chaîne.

Nous appelons une telle boucle un "*cycle de causalité*" (boucle combinatoire) s'il est effectivement infini en temps. Nous rajoutons alors généralement l'hypothèse suivante : la structure est acyclique c'est à dire qu'il existe au moins un registre ou autre élément de mémorisation permettant de "couper" ces boucles afin de rendre possible l'évaluation

NB : Même s'il existe des constructions combinatoires cycliques qui sont correctes car "stables" : pour toute combinaison des entrées chaque évaluation est finie dans le temps. Ces dernières sont en général rejetées par la majorité des formalismes synchrones.

Nous allons maintenant présenter très succinctement Lustre, Signal et Esterel/SyncChart.

2.1.2 Lustre

Le langage le plus proche de cet "assembleur" est Lustre [42], où nous décrivons sous une forme SSA de manière déclarative un système, ces équations s'appliquent sur des "flots", elles sont ensuite triées et nous vérifions que le système est acyclique. Ce langage permet de décrire des "sous-horloges" (de typage) grâce à un opérateur nommé *when*. Il est aussi possible de sur-échantillonner une horloge par rapport à l'horloge "mère" au travers de l'opérateur *current*. Toute équation non typée explicitement a le type de l'horloge "mère" de l'arbre de typage. Dans le langage Lustre, la vérification des horloges est purement syntaxique. Il n'y a pas d'inférence de type contrairement à Lucid Synchrone [26], qui est une extension du langage Lustre combiné avec certaines spécificités des langages ML : fonctions de haut-niveau, inférence de types et d'horloges.

Exemple :

```
node Buf2 ( val_in, stop_out : bool ) --INPUTS
    returns ( val_out, stop_in, aux, main, error :bool ); --OUTPUTS
let
        main = false -> (val_in) or (pre(main) and stop_out)
                    or (pre(main) and pre(aux));
        aux = false ->  (pre(main) and val_in and stop_out)
                    or ( pre(aux) and stop_out);
        stop_in = false-> pre(aux);
        val_out =  false -> ( not stop_out and pre(main) and not pre(aux))
                      or ( not stop_out and pre(aux));
        error = false -> val_in and stop_in; --SHOULD ASSERT ALWAYS FALSE
tel;
```

Un programme Lustre est décomposé en un ou plusieurs *node* (équivalent de fonction ou module), avec la description des entrées (première série d'arguments)

et sorties après le mot clef *returns* (seconde série d'arguments). Le corps du *node* est défini entre la paire *let*/*tel*. À L'intérieur du corps, nous décrivons les équations à l'aide des opérateurs binaires usuels de la logique, de l'arithmétique ; de l'opérateur mémoire *pre* utilisé en général de concert avec l'opérateur → (suivis de) permettant de lui associer une valeur initiale.
Par exemple, l'équation "stop_in = false -> pre(aux)" signifie qu'à l'instant initial stop_in est au niveau bas (false), et les instants d'après il contient la valeur à l'instant précédent (pre) du signal aux.

2.1.3 Signal

Signal est comme son cousin Lustre un langage déclaratif sous forme SSA. Il permet de spécifier des constructions plus complexes au niveau des relations entre les horloges [1]. Dans le langage Lustre tous les sous-échantillonnages dérivent d'une même racine qui est l'horloge de base. Dans le langage Signal il est possible de dériver de plusieurs horloges de bases. Nous disons alors que Signal est "multi-horloge" : nous obtenons non plus un arbre de typage mais une forêt. Lors de l'implantation cette forêt sera transformée en un arbre en rajoutant des contraintes supplémentaires.

2.1.4 Esterel/SyncCharts

Dans Esterel et SyncCharts nous sommes face à deux langages impératifs qui n'ont pas du tout une forme SSA comme les deux précédents langages. Il va falloir transformer ces derniers sous forme équationnelle (SSA) et ordonnancer ces équations. Ces compilations sont complexes et pas forcément complètes [60, 38, 76, 64, 75]. Dans Signal et Lustre le parallélisme n'est pas explicite contrairement à Esterel et SyncCharts. Nous allons décrire plus en détail ces deux langages.

Ces deux langages introduisent le concept de *préemptions* : la préemption forte *strong abort*, la préemption faible *weak abort*, la suspension *suspend*. Au travers de ces différentes constructions il est possible de décrire des automates. Chacune de ces préemptions s'applique sur le corps d'un état (qui peut être une hiérarchie d'automates). La préemption forte arrête immédiatement l'exécution du corps. La préemption faible arrête l'exécution du corps en lui laissant effectuer en quelque sorte ses "dernières volontés" pour l'instant courant. La suspension interdit l'exécution du corps pour l'instant courant.

Nous allons présenter un exemple : il s'agit d'un contrôleur pour un processeur pipeliné de 5 étages qui sont respectivement la lecture de l'instruction, le décodage de l'instruction, la lecture des opérandes, l'exécution et l'écriture des données

en parallèle avec la mise à jour du compteur ordinal. Un *module* est l'équivalent d'une fonction en Esterel. Il a un certain nombre d'entrées *input*, de sorties *output*, entrées/sorties *inputoutput* et des signaux locaux. Nous utilisons la construction de boucle infinie *loop ... end loop* pour le corps du module. À l'intérieur, nous avons une hiérarchie pour les resets *hardware* et *software*. Le reset hardware préempte toutes les opérations du pipeline. Par contre le reset software laisse exécuter les dernières volontées de l'instant courant avant d'effectuer le reset à proprement parlé ; pour ce faire nous utilisons d'abord la préemption forte "*abort ... when condition do ... end abort*" et à l'intérieur la préemption faible "*weak abort ... when condition do ... end abort*". Lorsque nous avons à la fois un reset hardware et software, le premier est plus prioritaire. Nous avons ensuite la description des 5 étages de pipelines fonctionnant en parallèle (opérateur ||, l'opérateur *;* signifiant en séquence) et dont la "synchronisation" entre ces étages est effectuée grâce à l'utilisation de l'opérateur *pre*, permettant la mémorisation de l'état d'activité de l'étage précédent lors de l'instant précédent, en utilisant la construction "*present condition then ... end present*". Chacun des étages est décrit aussi avec une boucle. Certains étages ne sont pas exécutés lorsque par exemple la mémoire est occupée à récupérer une donnée ou une instruction, ou en train d' écrire une donnée : pour ce faire nous utilisons la primitive de suspension "*suspend ... when condition*" avec l'utilisation du modificateur *immediat* sur la condition qui est alors prise en compte dès l'activation de l'instant (lors de l'initialisation par exemple). L'instruction *pause* indique la fin de la branche pour l'instant donné.

```
module proc:
%Pipeline hazard
input ReadBusy, WriteBusy;
%Scheduler output
output ReadInstruction, DecodeInstruction;
output ReadData, Compute;
output WriteData, UpdatePC;
output tempActivate;
inputoutput hardReset, softReset;
output init;

%Simple Processor with 5 Stage Pipeline
loop
 abort
  weak abort
   [ loop [suspend
      emit ReadInstruction
     when immediate ReadBusy]; pause end loop] || %First Stage ReadInstruction
   [ loop present pre(ReadInstruction)  then
      emit DecodeInstruction
     end present; pause end loop ] || %Second Stage DecodeInstruction
   [ loop present pre(DecodeInstruction) then
      suspend [emit ReadData] when immediate ReadBusy
     end present; pause end loop ] || %Third Stage ReadDatas
 [ loop present pre(ReadData) then
       emit Compute
     end present; pause end loop ] || %Fourth Stage Compute
   [ loop present pre(Compute) then [suspend
```

```
      [emit WriteData] || [emit UpdatePC]
      when immediate WriteBusy]
    end present; pause end loop ] %Last Stage - UpdatePC and Write Result
  when softReset do
    emit tempActivate; pause; emit init; pause
  end abort
 when hardReset do
    emit init; pause
 end abort
end loop
end module
```

Nous allons maintenant présenter sommairement le langage graphique SyncCharts qui partage les mêmes concepts qu'Esterel.

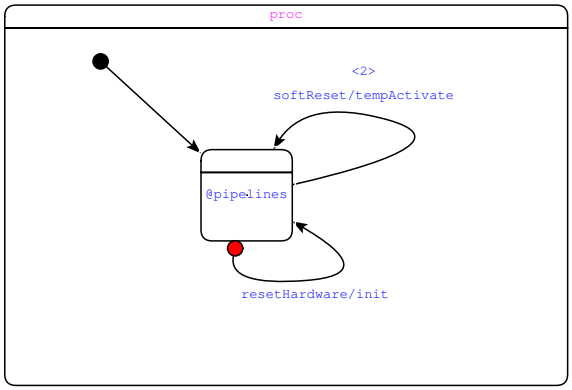


FIG. 2.1 – syncchart : exemple 1

Nous allons reprendre le précédent exemple. Dans le modèle SyncCharts, l'équivalent de la fonction (module) est un *macrostate* qui est la boîte rectangulaire à coins arrondis englobante nommée "proc" dans la figure 2.1. Chaque macrostate contient un ou plusieurs *STGs (State Transition Graphs)* qui fonctionnent en parallèle. Ces derniers sont séparés par des lignes pointillées comme l'illustre la figure 2.2. Nous exécutons en parallèle les cinq étages du pipeline.

Chaque STG de nos étages est constitué par un ensemble d'états (cercles), macrostates, pseudo-états initiaux (petits cercles noirs), et des arcs de préemptions : préemption faible notée par un arc simple, préemption forte avec un arc ayant à sa source un petit cercle, la suspension qui lie un petit cercle sur un état. Le symbole # symbolise le caractère *immédiat* de la transition. Il est possible d'émettre des signaux lors des transitions, mais aussi sur l'état (appelé *sustain*) noté */ Signal1, Signal2,...*. Tous les arcs sortant d'un état ou macrostate sont ordonnés suivant un ordre total par le développeur. (Par exemple sur la figure 2.1, l'arc de préemption forte *resetHardware* de priorité implicite $< 1 >$ est plus prioritaire que l'arc de

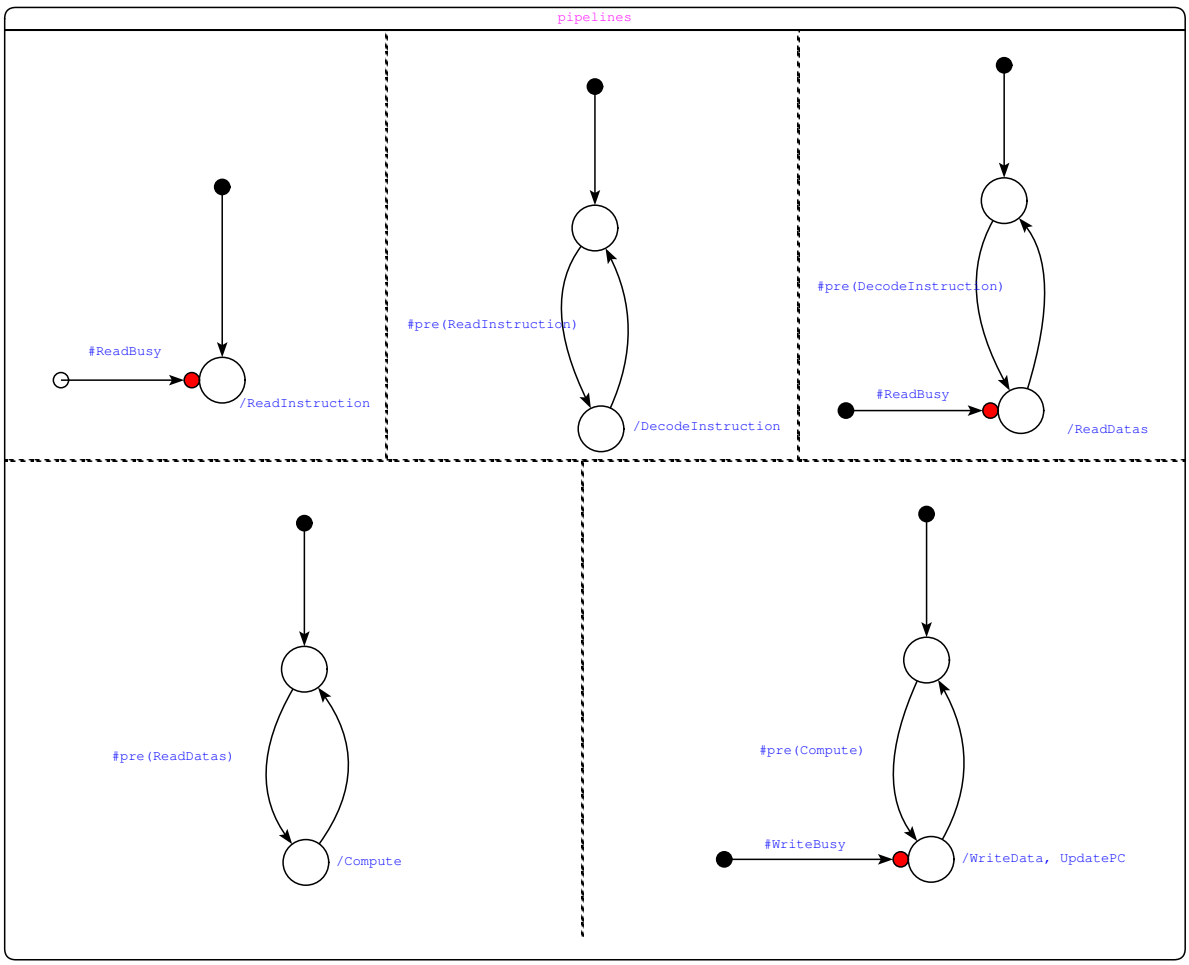


FIG. 2.2 – syncchart : exemple 2

préemption faible *softReset* de priorité $< 2 >$).

Pour comprendre plus en détail les différents styles de programmation de langages synchrones, le lecteur est invité à lire [6, 9, 42, 17].

Les langages synchrones ont maintenant largement dépassé le cadre de leur domaine d'application initial. Ils sont utilisés aussi dans la conception de puces, avionique, automobile...etc...

2.1.5 Résumé

Dans cette section, nous avons introduit les modèles synchrones.

Le modèle synchrone est un modèle où le parallélisme est déterministe, grâce à la notion d'instant cadencé par une horloge ; tous les signaux sont présents ou absents à chaque instant, les mémoires (registres) sont toujours "pleines" à chaque coup d'horloge.

Les hypothèses du modèle synchrone sont les suivantes :

- échantillonnage parfait : à tout endroit du système nous disposons des mêmes valeurs pour les signaux. Les signaux prennent une et une seule valeur dans l'instant amenant à une propriété de déterminisme.
- temps de calcul nul.
- temps de communication nul.
- absence de cycle de causalité : il existe au moins un registre coupant chaque cycle.
- pas d'hypothèse remise en cause au sein de l'exécution : contre-exemple si un signal est absent alors nous l'émettons.
- existe une horloge logique globale ou une forêt d'horloges logiques dans le cas du langage Signal, l'"absence" (logique) est codée par une valeur supplémentaire.

Le modèle synchrone est bien fondé et formalisé mathématiquement permettant d'effectuer des preuves et vérifications formelles.

Le modèle synchrone est un cas particulier d'une sous-classe bien connue des réseaux de Petri : les Marked Event Graphs.

2.2 Marked Event Graphs et SDF

Proverbe : Qui ne fait pas quand il peut ne fait pas quand il veut.

Les Marked Event Graphs, et ses extensions TMG (Timed Marked Event Graph) et SDF (Synchronous Data Flow) sont des sous classes particulières de réseaux de Petri qui ont la particularité de ne pas avoir de conflit. Ces modèles sont déterministes et confluents : ce qui signifie que tous les ordres d'exécutions des transitions "tirables" finissent par autoriser les mêmes comportements, juste plus ou moins décalés dans le temps (sous l'hypothèse d'équité que toute transition tirable sera éventuellement exécutée). Nous allons décrire en détail les Marked Event Graphs, nous présenterons ensuite la sémantique ASAP (As Soon As Possible) qui procède (possiblement simultanément) à tous les tirages possibles de transitions à chaque étape (c'est l'exécution la plus "rapide" du Marked Event Graph). Nous décrirons ensuite les extensions TMG et SDF qui introduisent des poids du type latence ou nombre de jetons respectivemement aux Marked Event Graphs.

2.2.1 Marked Event Graphs

Maintenant, nous allons décrire une classe particulière de réseaux de Petri que l'on dénomme les Marked Event Graphs (MG) (ou quelquefois appelés Event Graphs dans la littérature) étudiée par F.Commoner, A.W. Holt, S.Even et A.Pnueli [34] (la première étude a été effectuée par H. Genrich [40]).

Dans le modèle Marked Event Graph, une place a exactement une transition en entrée et une en sortie. Cette simplification structurelle permet de prouver des propriétés usuellement indécidables sur les réseaux de Petri généraux : par exemple le problème de la terminaison.

Dans les Marked Event Graphs, toute place a une seule entrée/sortie : une place peut alors être vue comme un *buffer* de jetons. Cette abstraction n'a aucun conflit/choix (*free-choice*) entre les comportements. Nous avons la possibilité de différer une activation d'une transition sans perdre l'activabilité.

Marquages Vivants et Sûrs d'un Graphe Orienté

Supposons que nous avons un graphe orienté fini $G(V,E)$, où V est l'ensemble des sommets (transitions) et E l'ensemble des arcs (places). La notation $e = a \rightarrow b$ signifie que l'arc e vient du sommet a et entre dans le sommet b. Nous affectons un nombre $M(e)$ de *jetons* (un nombre non négatif) à chaque arc e. La fonction M est appelée un *marquage* du graphe. Un sommet est dit *exécutable* si le nombre de jetons sur chacun des arcs entrants est strictement positif. L'exécution d'un sommet exécutable consiste à prendre un jeton sur chacun des arcs entrants, et d'ajouter un

jeton à chacune de ses arcs sortants. Puisque le nombre d'arcs entrants et sortants n'est pas forcément le même, le nombre total de jetons sur le graphe peut augmenter ou diminuer par le jeu des exécutions. Nous pouvons considérer le nombre de jetons sur un circuit simple orienté C ; ce nombre, $< M|C >$, est la somme des jetons portés par les arcs du circuit. Le lemme suivant est une conséquence directe de la définition de l'exécution.

Lemme 1. *Le nombre de jetons dans un circuit ne change pas lors de l'exécution.*

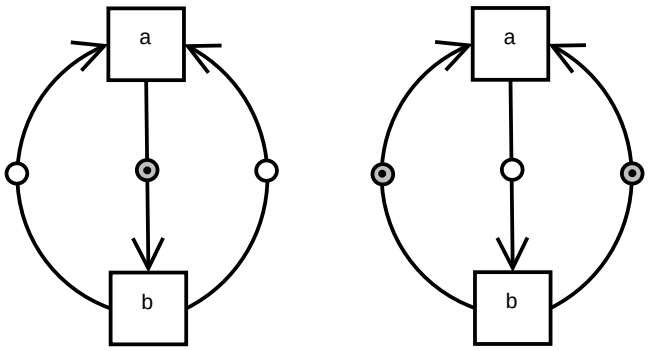


FIG. 2.3 – Marked Event Graphs : exemple 1

Exemple 1 Un exemple de Marked Event Graph est montré dans la figure précédente. Les sommets sont représentés par des carrés. Les ronds à l'intérieur des cercles permettent de visualiser le nombre de jetons sur chaque arc à un instant donné. Il n'y a qu'un seul jeton dans le marquage, et il est sur l'arc allant du sommet *a* vers le sommet *b*. Le sommet *b* est exécutable. Après exécution du sommet *b*, les deux arcs qui vont de *b* vers *a* ont chacun un jeton et donc *a* est exécutable. Après exécution de *a* nous retournons sur le marquage initial. Ainsi, le nombre de jetons dans le graphe change de 1 vers 2 et revient à 1 ; mais le nombre de jetons sur chacun des circuits élémentaires reste à 1.

Un marquage est dit *vivant* si tout sommet est exécutable, ou peut le devenir à travers une séquence d'exécutions.

Nous ajoutons l'hypothèse suivante : le graphe est une partie fortement connexe. Toute transition appartient à au moins un circuit élémentaire.

Théorème 2 (Vivacité Marked Event Graph)**.** *Un marquage est vivant si et seulement si le nombre de jetons sur tout circuit est strictement positif.*

Démonstration. Si le nombre de jetons sur un circuit est zéro, alors aucun sommet n'est exécutable dans ce circuit ; puisque le nombre de jetons ne change pas même

si d'autres arcs sont exécutés alors par le lemme 1 aucun sommet de ce circuit ne peut devenir exécutable par une séquence d'exécutions.

Maintenant supposons que le nombre de jetons pour tout circuit soit positif. Soit υ n'importe quel sommet du graphe. Considérons les arcs entrants à υ qui ne disposent pas de jeton. S'il n'y en a aucun alors le sommet est exécutable. Si ce n'est pas le cas, considérons les sommets d'où ces arcs arrivent. Si chacun d'entre eux est exécutable immédiatement, alors, clairement, υ deviendra exécutable après que chacun d'entre eux soit exécuté. Si quelques uns ne le sont pas, alors nous considérons les arcs entrants ne disposant pas de jetons, etc. Au fur et à mesure que nous continuons cette preuve par induction sur le chemin ("backtracking"), nous sélectionnons un sous-graphe de G qui consiste en υ, les arcs entrants en υ ne disposant pas de jeton, les sommets d'où ses arcs viennent, les arcs entrants à ces derniers, etc. Ce processus doit se terminer, puisque G est fini. Maintenant, ce sous-graphe doit avoir au moins un sommet qui doit ne pas avoir d'arc entrant émanant du sous-graphe. Ce sommet est exécutable dans le marquage courant de G. Après l'avoir exécuté, le sous-graphe du backtracking depuis υ est réduit d'un sommet. En répétant ce processus, on peut rendre exécutable υ. □

Corollaire 3. *Vivant je suis, vivant je reste : Un Marked Event Graph qui est vivant le reste toujours après exécution.*

Démonstration. Puisque le nombre de jetons dans les circuits est invariant par exécution (confère lemme 1), et que son marquage est vivant, alors le nombre de jetons sur tous les circuits est positif par le théorème 2, et ce nombre restera positif après exécution. Par le théorème 2 le marquage restera donc vivant. □

Un marquage est dit *sûr* si aucun arc a plus d'un jeton, et si aucune séquence d'exécution peut mettre plus de deux jetons ou plus sur un seul arc.

Théorème 4. *Un marquage vivant est sûr si et seulement si tout arc dans le graphe est dans un circuit avec un nombre de jetons égal à 1.*

Démonstration. Si pour toutes les arcs il existe un circuit dans lequel cet arc appartient, avec exactement un jeton dessus, alors par le lemme 1 le nombre de jetons sur ce circuit restera à 1, et donc il n'y aura jamais deux jetons ou plus sur cet arc.

Supposons qu'il existe un arc e, $a \rightarrow b$, telle que tous les circuits qui passent à travers ce dernier ont un nombre de jetons égal ou supérieur à 2. Nous voulons démontrer que par une séquence donnée d'exécutions nous pouvons placer deux jetons sur e. S'il n'y a pas de jeton sur e, alors nous "backtrackons" sur le sous-graphe où les arcs n'ont pas de jeton en partant du sommet a, comme dans la preuve du théorème 2. De la même manière que le théorème 2, nous pouvons rendre le sommet a exécutable et l'exécuter. Nous plaçons donc un jeton sur e.

Nous répétons la construction. Pareillement, le sous-graphe où les arcs n'ont pas de jeton depuis *a* n'inclut pas *b*, puisque cela impliquerait l'existence d'un circuit dont le nombre de jetons est égal à 1 passe par *e*. Alors, nous pouvons exécuter le sommet *a* sans exécuter le sommet *b*, et donc rajouter un second jeton sur *e*. Ainsi, le marquage initial n'est pas sûr. □

Corollaire 5. *Si un graphe dispose d'un marquage vivant et sûr, alors pour tout arc du graphe il existe un circuit qui passe par cette arc.*

C'est une conséquence immédiate du théorème 4.

2.2.2 Marked Event Graph avec places à capacité bornée

Définition 2 (Marked Event Graph à capacité infinie/bornée)**.** Si nous supposons que chaque place peut potentiellement contenir un nombre illimité de jetons, alors nous appelons un tel MG : un MG à *capacité infinie*. Inversement si un MG impose un nombre fini de jetons dans une place alors nous appelons un tel réseau un MG à *capacité bornée*. Dans le cas d'un MG à capacité bornée nous associons à chaque place une capacité : nous ajoutons un paramètre K en plus dans le n-uplet décrivant un Marked Event Graph, que nous notons : $MG_K = (P, T, A, M_0, K)$ où P est l'ensemble des places, T est l'ensemble des transitions, A est l'ensemble des arcs tel que $A \subseteq (PxT) \cup (TxP)$, M_0 est le marquage initial des places $M_0 : P \rightarrow \mathbb{N}$ et finalement $K : P \rightarrow \mathbb{N}^+$ est une restriction sur la capacité des places, nous associons à chaque place $s \in P$ un entier positif $n \in \mathbb{N}^+$ qui décrit le nombre maximum de jetons que la place peut contenir.

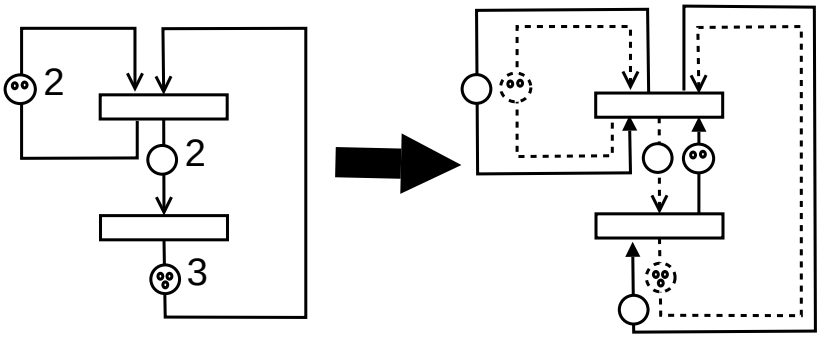


FIG. 2.4 – Transformation MG à capacité bornée vers MG à capacité non-bornée

Transformation MG à capacité bornée vers un MG à capacité non bornée La transformation d'un Marked Event Graph à capacité bornée vers un Marked Event Graph non-borné est effectuée de la manière suivante : à toute place p de capacité $K(p)$ reliée par un arc entre deux transitions $(Source, Destination)$, nous ajoutons un autre arc avec une place p' orienté dans l'autre sens $(Destination, Source)$ avec $K(p) - M_0(p)$ jetons où $M_0(p)$ est le nombre de jetons présents initialement dans la première place.

Le fait d'avoir ajouté ces places créé un mécanisme de contrôle de flot (*back-pressure*). Nous avons en fait créé des circuits qui satisfont l'invariant de capacité $K(p)$ associé à chaque place : $K(p') = K(p) = p + p'$. Cette transformation et sa preuve formelle sont généralement utilisées implicitement dans la littérature des réseaux de Petri. Une preuve formelle d'équivalence de comportement est disponible dans le papier suivant [3].

2.2.3 Extensions

Timed Marked Event Graph

Timed Marked Event Graph est un modèle qui a été introduit dans la thèse de Chander Ramchandani [65] en 1974. Il s'agit d'une extension du modèle Marked Event Graphs où les transitions sont annotées par un nombre entier représentant le temps nécessaire pour que les jetons présents sur toutes les entrées atteignent toutes les sorties lorsque la transition est exécutée confère figure 2.5 (a). Ramchandani a montré tout d'abord qu'il pouvait transformer le précédent modèle vers un Marked Event Graph où la latence de calcul est dirigée des transitions vers les places illustré sur la figure 2.5 (b). Grâce à une expansion des latences sur les places en introduisant de nouvelles transitions de "transport" (associées avec des places de latence unitaire), il obtient alors un Marked Event Graph montré dans la figure 2.5 (c). Ce Marked Event Graph permet de déterminer de manière précise la progression des jetons à chaque instant, donnant ainsi une sémantique qui était absente du modèle TMG avec les places annotées par la latence.

Sémantique ASAP

Définition 3 (ASAP)**.** Si toutes les transitions s'exécutent en suivant la règle *au plus tôt* (sémantique As Soon As Possible (ASAP)), alors elles s'exécutent dès qu'il y a suffisamment de jetons sur toutes leurs entrées.

Employer cette règle d'exécution ou toute autre n'altère en aucun des cas l'ordre partiel des évènements dans les Marked Event Graphs. Chaque transition

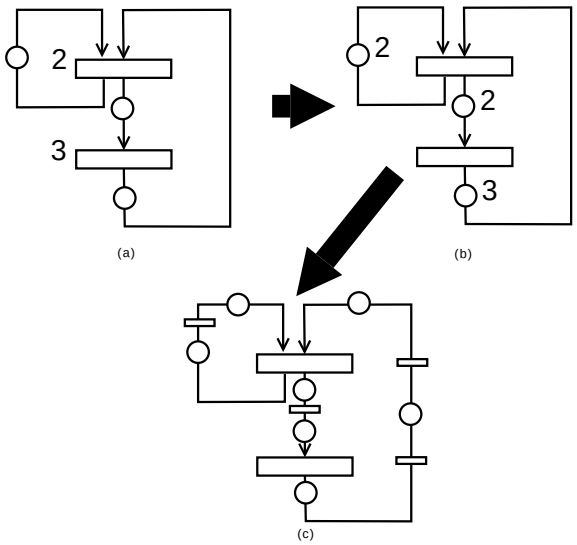


FIG. 2.5 – Transformation TMG vers MG

ne peut s'exécuter que si elle dispose d'au moins un jeton sur chacune de ses entrées. La sémantique ASAP a la propriété de donner la date au plus tôt (instant) d'exécution de chaque transition.

Synchronous Data Flow

Synchronous Data Flow (SDF) [53, 52] est un raffinement des Marked Event Graphs introduit par Edward A. Lee et David G. Messerschmitt dans lequel on associe des nombres de jetons (des poids) consommés/produits aux entrées/sorties des transitions. Un graphe SDF est dit *homogène* et équivalent à un *Marked Event Graph* : si chaque entrée/sortie de toute transition du graphe est annotée du même poids.

SDF permet de décrire des systèmes où nous voulons trouver une solution afin d'obtenir un ordonnancement statique périodique efficace dans le cas de composants synchrones fonctionnant à la même fréquence d'horloge mais disposant d'échantillonnages différents (ou largeurs de bus).

Formellement un graphe SDF est un graphe orienté fini $G(V,E)$ où V est l'ensemble des sommets, E l'ensemble des arcs. La notation $e = a \rightarrow b$ signifie que l'arc e vient du sommet a et entre dans le sommet b. Nous affectons un nombre $M(e)$ de jetons (non négatif) à chaque arc e. Cette fonction est appelée marquage. À chaque sommet nous associons un ensemble d'entrées et sorties, qui sont toutes annotées par un nombre strictement positif (poids). Un sommet est dit exécutable si le nombre de jetons sur chacun des arc entrants est égal ou supérieur au poids

associé à l'entrée correspondante. Une exécution d'un sommet ajoute out_i jetons à chaque sortie et retire in_j jetons à chaque entrée où out_i et in_j sont les poids associés à une sortie i du sommet et respectivement à une entrée j du sommet.

Le problème principal dans ce modèle est d'identifier *l'inconsistence* entre les taux de production/consommation qui demandent des ressources infinies de stockage temporaire.

Un graphe SDF peut être caractérisé par une matrice, où nous affectons une colonne à chaque sommet et une ligne à chaque arc. La (i, j)ième entrée dans la matrice est la quantité de données produites et consommées par un sommet j sur l'arc i, la valeur consommée est négative et similairement produite est positive. Nous appellerons une telle matrice : *matrice topologique*. Si un sommet a une boucle sur lui-même alors seulement une entrée dans la matrice illustrera un tel lien. Cette entrée donne la différence entre la quantité de données produites et consommées à chaque fois par ce sommet.

Lee a montré dans [52, 53] qu'une condition nécessaire pour l'existence d'un tel ordonnancement est que le *rang* de la matrice topologique τ est $rang(\tau) = s-1$ où s est le nombre de sommets dans la matrice.

Même si les taux de production/consommation sont consistants, il peut être impossible de construire un ordonnancement périodique admissible. Par exemple, les graphes cycliques ont besoin d'un certain nombre de jetons initiaux afin d'être exécutables.

Malheureusement contrairement aux Marked Event Graphs, déterminer analytiquement si une partie fortement connexe est vivante est un problème difficile dans le cas général de SDF.

Nous allons introduire maintenant un ensemble de lemmes, corollaires et théorèmes qui sont issus de [52] et qui nous permettent de définir un graphe SDF *sound* qui est un graphe SDF qui a des buffers de taille bornée. Les preuves sont fournies dans l'annexe B.

Lemme 6. *Toutes les matrices topologiques pour un graphe SDF donné ont le même rang.*

Lemme 7. *Une matrice topologique pour un arbre à un rang $s-1$ où s est le nombre de sommets (un arbre est un graphe connexe sans cycle, où nous ignorons la direction des arcs).*

Lemme 8. *Pour un graphe SDF connexe avec une matrice topologique τ : $rang(\tau) \geq s-1$ où s est le nombre de sommets dans le graphe.*

Corollaire 9. *Le rang d'une matrice topologique est $s-1$ ou s.*

Un *ordonnancement séquentiel admissible* ϕ est une liste ordonnée non-vide de sommets telle que si les sommets sont exécutés dans la séquence donnée par ϕ,

la quantité de données dans les buffers sera a tout instant non-négative et bornée. Chaque nœud doit apparaître dans ϕ au moins une fois. Un *ordonnancement séquentiel périodique admissible* (Periodic Admissible Sequential Schedule - PASS) est un ordonnancement séquentiel périodique et infini admissible. Il est spécifié par une liste ϕ qui est la liste des sommets dans une période.

Théorème 10. *Pour un graphe connexe SDF avec s sommets et une matrice topologique* τ, $rang(\tau) = s-1$ *est une condition nécessaire pour qu'un ordonnancement séquentiel périodique admissible existe.*

Lemme 11. *Supposons un graphe SDF connexe avec une matrice topologique* τ. *Soit q n'importe quel vecteur tel que* $\tau q = O$. *Prenons un chemin connexe passant au travers du graphe par l'ensemble* $B = \{b_1, ..., b_L\}$ *où chaque entrée désigne un sommet, et le sommet* b_1 *est connexe au sommet* b_2*,..., jusqu'au sommet* b_L. *Alors tous les* q_i, $i \in B$ *sont des zéros, ou sont tous strictement positifs, ou sont tous strictement négatifs. De plus, si n'importe quel* q_i *est rationel alors tous les* q_i *sont rationnels.*

Théorème 12. *Pour un graphe SDF connexe avec s sommets et une matrice topologique* τ, *avec* $rang(\tau) = s-1$, *nous pouvons trouver un vecteur d'entiers positifs* $q \neq O$ *tel que* $\tau q = O$ *où O est le vecteur nul.*

Un graphe SDF *sound* est un graphe SDF qui satisfait la condition de rang et dispose d'un ordonnancement périodique admissible.

Graphes SDF *uniformes* Maintenant nous allons définir une classe de graphes SDF particuliers que nous dénommons graphes SDF *uniformes*. Ces graphes ont la particularité que chaque sommet a localement le même "poids" sur toutes ses entrées et toutes ses sorties. Ces graphes sont intéressants car ils ont la propriété de toujours avoir un ordonnancement où la taille des buffers est toujours bornée. Il nous est aussi possible de définir quelques bornes sur le nombre de jetons présents sur chaque circuit élémentaire afin de disposer d'un système *vivant*.

Théorème 13. *Tout graphe SDF* uniforme *est* sound, *i.e. il existe un ordonnancement périodique avec des buffers de taille bornée.*

Démonstration. Chaque sommet produit/consomme uniformément à chaque exécution, il suffit de prendre alors le Plus Petit Commun Multiple (PPCM) des entrées ou des sorties qui sont strictement égaux par définition ; puis de diviser ce PPCM par le nombre porté sur chaque sommet, nous obtenons alors le nombre d'exécutions du sommet pour l'ensemble de la période. □

Lemme 14. *Le PPCM(nombres)/nombre du graphe SDF* uniforme *donne pour chaque sommet dans le graphe le nombre d'exécutions minimum pour chacun de ces sommets.*

Démonstration. Trivial, il n'existe pas de période plus courte que le PPCM. □

Lemme 15. *Sur chaque circuit élémentaire du graphe SDF* uniforme *le nombre de jetons est invariant.*

Démonstration. Voir la preuve correspondante pour le lemme 1 dans la section Marked Event Graph, la différence est qu'au lieu de consommer et produire un jeton, nous en consommons le même nombre en entrée et produisons le même en sortie. □

Nous allons maintenant décrire deux bornes hautes permettant d'assurer la vivacité, toutes deux sont des conditions suffisantes, pas nécessaires.

Théorème 16 (Borne vivacité haute (synchrone))**.** *Tout graphe SDF* uniforme *est vivant si sur chaque circuit élémentaire il y a n jetons sur chaque entrée de sommets où n est l'entier porté par le sommet en question.* NB :*Ceci est assimilable à une exécution "synchrone".*

Démonstration. Tous les sommets disposent d'assez de jetons sur toutes leurs entrées, ils sont tous exécutables initialement. Or le lemme précédent dit que le nombre de jetons est invariant sur chaque circuit élémentaire, il en résulte qu'un tel graphe est vivant.

Nous employons la même technique de backtracking que dans la preuve de la vivacité des Marked Event Graph. Ce qui va se passer est qu'il existe toujours au moins un sommet qui peut toujours s'exécuter : si un nœud produit plus ou autant que nécessaire à son successeur, alors pas de soucis, sinon il existe forcément l'un de ces sommets qui peut s'exécuter par un backtracking, grâce au précédent lemme. □

Il faut donc $\Sigma_{i=1}^{k} n_i$ jetons initiaux dans chaque circuit élémentaire, n_i est le nombre associé à chaque sommet d'indice i du circuit en question contenant k sommets. Sur la figure 2.6 (a), nous décrivons un circuit avec $7+3+2=12$ jetons.

Nous allons décrire maintenant une autre borne maximale qui est "plus petite" au niveau du nombre de jetons nécessaires lors de l'initialisation.

Théorème 17 (Borne vivacité haute (par passage de relais))**.** *Tout graphe SDF* uniforme *est vivant si sur chaque circuit élémentaire il y a n jetons sur un des sommets et sur tous les autres il y a* $n-1$ *jetons.*

Démonstration. Pareillement nous allons nous appuyer sur le lemme d'invariance du nombre de jetons sur chaque circuit élémentaire. Le schéma de la preuve est le même que précédemment. En construisant l'arbre de backtracking pour chaque circuit élémentaire, il existe un sommet qui est exécutable. Le système est donc vivant. □

Il faut donc $(\Sigma_{i=1}^{k-1} n_i - 1) + n_k$ jetons initiaux dans chaque circuit élémentaire, n_i est le poids associé à chaque sommet i du circuit contenant k sommets. Sur la figure 2.6 (b), nous décrivons un circuit avec $(7-1)+(2-1)+3 = 10$ jetons.

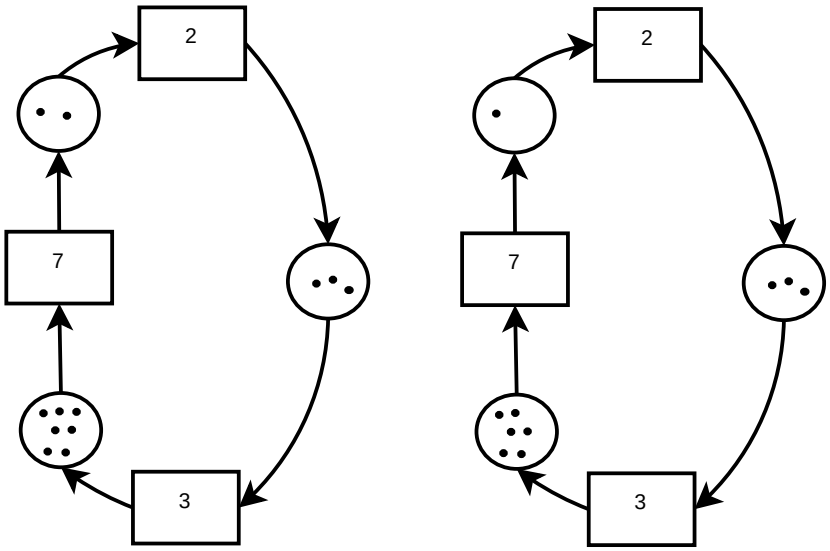


FIG. 2.6 – SDF uniforme : (a) marquage synchrone (b) marquage par passage de relais

2.3 Relations entre les modèles synchrones et asynchrones

Il importe en peinture, que le portrait ressemble au modèle, mais non pas le modèle au portrait. Paul-Jean Toulet

Dans cette sous-section, nous souhaitons illustrer quels sont les liens entre les modèles synchrones et asynchrones que nous avons décrits jusqu'à présent. Nous allons montrer que le modèle Latency Insensitive est une implantation particulière des Marked Event Graphs dits ASAP dont la capacité des places est bornée à une capacité de 2. Pour reprendre la citation précédente nous souhaitons dépeindre que finalement le *"modèle Latency Insensitive ressemble à une classe particulière du portrait Marked Event Graphs"*.

La figure 2.7 montre ces liens. Nous nous ramenons au modèle des Marked Event Graphs utilisant une sémantique ASAP.

Nous allons d'abord discuter de l'importance de l'absence de conflit et la confluence dans ces modèles. L'absence de conflit assurre une partie du déterminisme de ces modèles, l'autre partie du déterminisme est issue des règles d'évaluation des nœuds de calculs (composants synchrones et transitions dans les cas asynchrones). La confluence dit que nous conservons les potentialités de franchissement des nœuds de calcul dans le futur : ainsi dans le cas des Marked Event Graphs quelque soit la règle d'évaluation des nœuds l'ordre des évènements sera conservé si nous abstrayons les "temps morts" (ou le système ne progresse pas). La règle d'évaluation ASAP donne la date au plus tôt à laquelle peuvent s'évaluer les nœuds de calcul. Contrairement aux réseaux de Petri généraux où une telle règle restreint l'ordre des évènements, la confluence dans nos modèles sans conflit assure que le fait d'être ASAP ne restreint pas cet ordre. Ceci nous permet de lier les Marked Event Graphs "asynchrones" (sans règle d'évaluation) avec les Marked Event Graphs ASAP. Nous pouvons aussi faire une abstraction du modèle synchrone vers les Synchronous Marked Event Graphs qui sont une sous-classe particulière des Marked Event Graphs ASAP où il n'y a pas de place sans jeton, ainsi à tout instant chaque transition dispose de tous les jetons nécessaires à son exécution, simulant ainsi un fonctionnement synchrone.

Nous avons vu auparavant que les Marked Event Graphs avec places à capacité bornée et les Timed Marked Event Graphs peuvent être transformés vers le modèle Marked Event Graphs "asynchrone" (c'est à dire sans règle d'exécution spécifiée) et Marked Event Graphs ASAP respectivement.

Dans le cas des Marked Event Graphs à capacité bornée, grâce à une transformation structurelle, nous obtenons un Marked Event Graphs "asynchrone" équivalent : en ajoutant des arcs supplémentaires créant des cycles qui satisfont l'in-

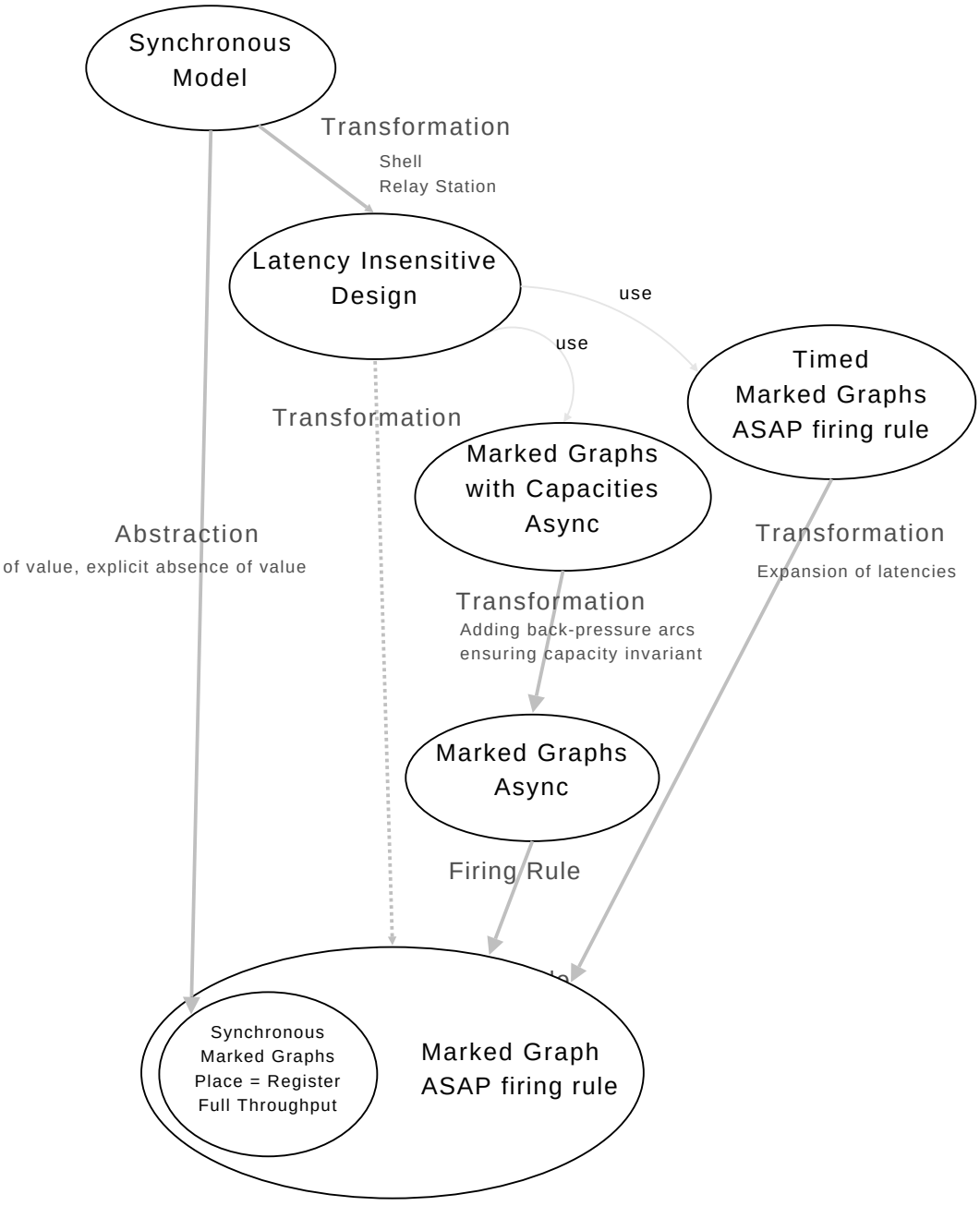


FIG. 2.7 – Relations entre modèles synchrones et asynchrones

variant de capacité de la spécification annotée avec ces capacités.

Dans le cas des Timed Marked Event Graphs au travers d'une expansion structurelle des latences en ajoutant des transitions de "transport" et en normalisant les latences des places à 1, nous obtenons alors un Marked Event Graph ASAP avec le même comportement.

Conditions initiales

Nous considérons ici différents problèmes d'initialisation et de structure "bien formée" d'un réseau, qui permettent de s'assurer pour chaque sémantique qu'elle soit synchrone ou asynchrone : à la fois l'absence de famine (ou vivacité) et l'absence de congestion (ou sûreté). Nous considérons aussi les problèmes quantitatifs sur les taux de production (c'est à dire les débits). Nous rappelons le fait bien connu que n'importe quel graphe peut être décomposé en un graphe acyclique orienté (DAG) de composantes fortement connexes (chacune d'entre elles peut contenir un seul nœud). Nous supposons que les entrées/sorties ne rajoutent pas de contraintes sur le système, c'est à dire que lorsque nous décrivons des cycles, ces derniers ont un débit propre et les entrées/sorties sont présentes quand nécessaires. Elles sont respectivement des pré-conditions/post-conditions.

Réseau synchrone Concernant les réseaux synchrones de composants synchrones, un signal/donnée valide doit être présent sur chaque fil au coup d'horloge. En général nous imposons qu'il n'y ait pas de boucle combinatoire dans le réseau. En d'autres termes *chaque circuit dans le réseau doit au moins avoir un registre,* qui produit sa sortie le prochain cycle d'horloge tout en recevant sa nouvelle entrée. Ici le réseau consiste en des dépendances locales entre les composants plus l'interconnection entre ces composants. C'est une condition strictement plus faible que d'imposer que toutes les sorties des composants soient *bufferisées* (comme dans le genre de Moore), même si cette supposition est généralement recommandée pour le style de design compositionnel, et en fait implicitement adoptée dans certaines lectures sur les GALS.

Réseau asynchrone Concernant les réseaux asynchrones de composants synchrones, c'est aussi le cas que le réseau est vivace (de telle manière que tous les composants locaux sont exécutés infiniment souvent) si et seulement s'il existe au moins un jeton dans chaque circuit (supposant que les entrées sont des flots infinis de signaux/données). C'est une conséquence directe de la vivacité des Marked Event Graphs. Cela correspond à l'hypothèse sur les réseaux synchrones, supposant que le registre est en fait un registre sur une sortie locale (mais comme auparavant il n'est pas nécessaire que toute sortie soit bufferisée, seulement au moins

une dans un cycle). Les sorties bufferisées peuvent être en ce sens vues depuis les composants locaux comme les “graines” (jetons portant les valeurs initiales des FIFOs d’interconnection). Bien sûr l’initialisation avec plus de valeurs dans la file est possible tout en préservant la vivacité (plus de jetons, plus de débit dans ce cas). Mais il est problématique de savoir comment obtenir ces graines (valeurs) en général si nous partons d’une spécification synchrone avec laquelle nous voulons conserver une équivalence fonctionnelle.

Réseau LID La théorie du Latency Insensitive Design (LID) initiée par Carloni (décrite en détail dans l’annexe A) permet de décrire des systèmes synchrones insensibles aux latences (des fils) à partir d’une spécification synchrone idéale : l’idée est d’encapsuler les nœuds de calcul synchrones dans des *Shells*, les communications sont assurées par des liaisons point à point grâce à des lignes de *Stations de Relais* de taille proportionnelle au nombre de cycles d’horloges nécessaires pour convoyer les données entre les *Shells*. La composition des *Shells* et *Stations de Relais* implante un protocole de contrôle de flot qui permet d’assurrer un critère de correction entre la spécification synchrone idéale et l’implémentation LID. Ce critère de correction est une équivalence modulo latence : l’ordre partiel des évènements du LID est compatible avec celui de la spécification synchrone.

Nous prenons comme définition abstraite des systèmes LID des Marked Event Graphs (expansés) avec des places de capacité bornée à 2. Ceci est justifié dans le chapitre suivant. D’autre part, nous fournissons une étude bibliographique des principales contributions historiques sur le LID en annexe A. Dans le manuscrit nous les appelerons des systèmes *synchrones relaxés* car ils combinent à la fois des caractéristiques synchrones (tous les composants et interconnections fonctionnent sur la même horloge), et des contraintes de latences minimales imposées par l’utilisateur (une constante entière de latence pour chaque ligne qui transmet les signaux et données). Si les données sont toujours en transit, les nœuds de calcul sont arrêtés par les *shells* les encapsulant en utilisant un mécanisme de *clock-gating*. Pour respecter la taille bornée des tampons un protocole de contrôle de congestion est appliqué entre les *stations de relais* et les *shells*.

Considérant les versions synchrones relaxées (appelé aussi LID), où les canaux de communication de taille bornée remplacent les FIFOs à capacité non bornée, un nouveau type de problème de vivacité arrive. À cause des congestions potentielles, les blocs de calculs locaux peuvent maintenant être bloqués car les canaux de sortie ne sont pas prêts à accepter les résultats, qu’ils ne pourraient pas stocker sans dépassement de capacité. Ce problème est théoriquement résolu si le Marked Event Graph complété n’a aucun cycle vide. Ici la complétion du Marked Event Graph consiste à ajouter des places et des arcs en sens inverse qui jouent le rôle de capacité. En d’autres termes chaque cycle du graphe dans le graphe

complété doit contenir au moins un jeton sur une de ces places.

Abstraction Synchrone vers Synchronous Marked Event Graphs

Nous souhaitons exprimer sous quelles conditions un réseau synchrone transformé en un Marked Event Graph fournit un ordre partiel compatible des évènements avec le réseau synchrone : c'est à dire une équivalence de flot, ou encore nommée équivalence modulo latence dans le cas du LID.

Nous savons déjà que le modèle synchrone est un cas particulier du modèle Marked Event Graph utilisant la règle d'évaluation ASAP. Nous supposons que la spécification synchrone satisfait les critères de correction dont nous avons discutés pour les conditions initiales.

Nous effectuons la transformation suivante pour passer du réseau synchrone au Marked Event Graph qui est équivalent :

- à chaque registre nous lui associons une place,
- à chaque "bloc de base" combinatoire nous lui associons une transition : un "bloc de base" combinatoire est une suite (séquence) maximale de blocs combinatoires liés uniquement par des fils.

La transformation dispose au final du même ensemble de registres/places. Si le réseau synchrone était vivace alors le Marked Event Graph l'est aussi par construction, il y a une équivalence structurelle entre les marquages.

Nous savons aussi que le fait d'utiliser n'importe quelle règle d'évaluation dans le Marked Event Graph ne fera que "décaler l'exécution" par rapport à l'ordre des évènements de la règle d'évaluation ASAP. Générant finalement un ordre partiel des évènements compatible avec l'ordre partiel des évènements de la spécification synchrone. C'est à dire : en supprimant les "temps morts" (évènements non porteur d'information utile) qui sont présents uniquement dans le LID et dont la cause est l'introduction des latences.

Transformation de Latency Insensitive vers Marked Event Graphs

Une équivalence modulo latence a été introduite entre le modèle Synchrone et le modèle Latency Insensitive dans la théorie dénotationielle introduite par Carloni disponible dans l'annexe A.

Nous recréons l'équivalence de manière similaire à la section précédente en utilisant une transformation vers les Marked Event Graphs. Nous savons que le "Shell" est assimilable à une transition fonctionnant grosso-modo avec la règle d'évaluation ASAP : dès que toutes ses entrées sont prêtes alors il s'exécute pourvu que toutes les places en sortie ne soient pas pleines. Pour ajouter cette contrainte, nous nous inspirons de la transformation effectuée lorsque nous pas-

sons d'un Marked Event Graph à capacité vers un Marked Event Graph sans capacité.

La transformation du réseau synchrone au modèle Latency Insensitive sous la forme d'un Marked Event Graph s'exécutant sous une règle d'évaluation ASAP est la suivante :

- Chaque "bloc de base" de Shells est transformé en une transition.
- Chaque "bloc de base" (ou ligne) de Stations de Relais est transformé par le pattern décrit dans la figure 2.8 : il s'agit d'une expression régulière où la transition permet de modéliser le transport sur le fil lorsque la latence est strictement supérieure à 1. Les liens montants modélisent le protocole de contrôle de congestion.

Comme précédemment nous supposons que la spécification synchrone est correcte, donc dans chaque cycle il existe au moins une valeur initiale permettant ainsi d'assurer la vivacité dans le Marked Event Graph obtenu.

Le marquage initial des Stations de Relais est effectué de la manière suivante : s'il n'y a pas de valeur initialement alors la place de contrôle de flot (sur les liens montants à droite sur la figure 2.8) est initialisée avec deux jetons, autrement il y a un jeton sur chacune des places de la station de relais : nous avons un cycle entre le couple de places associé à chaque Station de Relais qui est de 2 jetons. Il en résulte que toutes les places du Marked Event Graph obtenu seront de capacité 2. Cette règle sur le marquage assure également la vivacité de l'ensemble du Marked Event Graph. Nous reviendrons en détail sur le fonctionnement interne d'une Station de Relais dans le prochain chapitre. Nous avons ajouté des transitions lors de la phase de transformation des "blocs de base" de Stations de Relais. Nous n'avons en fait que "rallonger" les chemins de données. Nous obtenons donc toujours un ordre partiel compatible avec celui de la spécification synchrone après abstraction des temps "morts".

Ainsi, cette transformation de LID vers Marked Event Graph possèdera la capacité de stocker $2n$ jetons sur une ligne de connection comptant n stations de relais.

2.4 Résumé

Dans cette section nous avons présenté les Marked Event Graphs et deux extensions TMG (Timed Marked Event Graphs) et SDF (Synchronous Data Flow).

Dans le cadre des Marked Event Graphs, le nombre de jetons consommés et produits est exactement 1, et les places ont strictement une entrée et une sortie, amenant l'ensemble de ce modèle à l'analyse grâce à l'absence de conflit. Nous avons rappelé les conditions nécessaires à la vivacité, l'initialisation dans le cadre de parties fortement connexes : il suffit de disposer d'un jeton sur chaque cycle

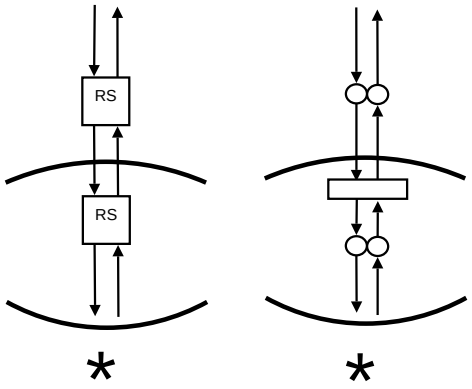


FIG. 2.8 – Transformation ligne de RS vers MG ASAP

élémentaire. Cette classe n'a pas de "choix" et est confluente : le fait d'exécuter une transition ne modifie en aucun cas les potentialités de franchissement d'autres transitions qui étaient exécutables.

Dans TMG, chaque transition est annotée par un entier représentant la latence de la transition. Il existe une transformation de ce modèle vers le modèle Marked Event Graph dit ASAP, via une expansion de la latence en introduisant de nouvelles transitions de "transport".

Dans SDF, le nombre de jetons consommés et produits est fixe et arbitraire. Il existe un problème d'accumulation/famine de jetons potentiel, une condition nécessaire pour éviter cela est que ces taux de production et consommation soient *équilibrés*, c'est à dire qu'il existe une séquence d'activation des différentes transitions de façon à ce que toutes les occurrences de jetons produites soient toutes consommées. La vivacité, l'initialisation dans le cadre de partie fortement connexe est généralement vérifiée par une phase de simulation atteignant un même état déja rencontré.

Nous avons introduit une classe particulière de classe SDF qualifiée d'*uniforme* car les nœuds consommment/produisent localement le même nombre de jetons. Nous avons montré pour cette classe de graphes SDF qu'il existe un ordonnancement périodique avec des buffers de capacité bornée, et nous avons donné une borne sur le nombre de jetons nécessaires dans chaque cycle élémentaire afin d'assurer la vivacité. Cette classe SDF *uniforme* permet de modéliser des systèmes multi-horloges où chaque transition est un système synchrone avec sa propre horloge (fréquence).

Le fait que la règle d'exécution soit ASAP dans le cas des Marked Event

Graphs, TMG et SDF n'altère pas la propriété de confluence.

Nous avons aussi décrit les liens existants entre les modèles Synchrone, Marked Event Graph non-borné, Marked Event Graph à capacité bornée et Latency Insensitive.

Un système synchrone est un système où à chaque instant il échantillonne toutes ses entrées, calcule et produit toutes ses sorties : il existe un ordre partiel des évènements entre les entrées, variables locales et les sorties générées. Nous pouvons transformer ce graphe synchrone en un Marked Event Graph : pour chaque registre nous associons une place et chaque bloc de combinatoire acyclique nous lui joignons une transition. Le Marked Event Graph peut exécuter une transition lorsque cette dernière dispose au moins d'un jeton sur chaque entrée. Nous parlons de règle d'exécution ASAP (As Soon As Possible) lorsqu'un Marked Event Graph exécute toute transition qui dispose d'un jeton exactement sur chacune de ses entrées. Le Marked Event Graph obtenu par la transformation précédente et la règle d'exécution ASAP a le même comportement que le système synchrone : ils ont tous les deux exactement le même ordre partiel des évènements, car toutes les places sont remplies. Nous appelons un tel Marked Event Graph ASAP un Synchronous Marked Event Graph.

Maintenant, partant du Synchronous Marked Event Graph précédent, nous insérons entre certaines transitions d'autres transitions qui ont la particularité d'avoir exactement une entrée et une sortie, nous leur allions aussi les places nécessaires entre elles. Nous avons en fait *raffiné* les liens de communication que nous avions supposés de latence nulle en des liens de latence fixe : tout ce qui va se passer est que nous allons en fait *décaler* sur un nombre arbitraire de cycles l'évaluation de certaines transitions. Nous obtenons alors un autre ordre partiel des évènements mais ce dernier est compatible avec le précédent : une réaction au lieu de prendre exactement un cycle va en prendre un nombre arbitraire mais borné.

Un système LID est une implémentation particulière d'un Marked Event Graph de capacité bornée à 2. Il s'agit d'un Marked Event Graph s'évaluant avec une règle ASAP, la transition est assimilée au *Shell* qui lorsque toutes ses entrées sont prêtes et qu'il y a suffisament de ressources afin de stocker toutes ses sorties alors il "s'exécute" (les places ne sont pas pleines) ; et une *station de relais* est assimilée à un couple de places montantes et descendantes disposant d'une capacité de 2 jetons. Dans LID tous les arcs reliant chaque entité entre elles (shells et stations de relais) ont exactement une latence d'un cycle (sauf shell vers shell). Or il se trouve que si une station de relais est reliée à deux sections formant un cycle élémentaire, il peut arriver la chose suivante : n'importe quelle station de relais reçoive en amont une donnée valide alors qu'en même temps elle ne puisse en-

voyer la donnée valide reçue le cycle précédent parcequ'il lui a été indiqué en aval qu'il y avait une congestion. Toute station de relais doit donc avoir au moins une capacité de 2 places dans ce cadre, pour assurrer le débit maximum de 1. L'autre partie du lien avec les Marked Event Graphs à capacité bornée vient aussi du fait que chaque "transition/place" a des liens permettant une signalisation amont/aval implémentant le protocole de contrôle de congestion et la construction de l'instant "global" par synchronisation.

Dans le prochain chapitre nous allons décrire plus en détail le modèle Latency Insensitive.

Chapitre 3

Modélisation du “Latency Insensitive Design”

Proverbe Chinois : On devrait gouverner un grand empire avec autant de simplicité que l'on fait cuire un petit poisson.

Ses latences induites par les “longs” fils introduisent des problèmes de “timing-closure” dans les systèmes sur puce modernes. La propagation des signaux au travers de l'ensemble de la puce en un seul cycle d'horloge est problématique. La théorie des systèmes dits insensibles à la latence (LID), proposée originellement par Luca Carloni, Kenneth McMillan et Alberto Sangiovanni-Vincentelli [62, 63] offre des solutions pour ce problème. La théorie peut grossièrement être décrite de la manière suivante : une spécification complètement synchrone (mono-instant) de référence est d'abord *“désynchronisée”* en un réseau asynchrone composé d'ilôts synchrones (un système GALS). Après des mécanismes d'interconnection sont introduits afin de *“resynchroniser”* le système global, mais en autorisant maintenant des latences spécifiques au niveau des interconnections (temps entiers), sous la forme de lignes de *stations de relais*. Les stations de relais avec les shells (“coquille”, conteneur) encapsulant chaque *“perle”* (blocs synchrones), sont chargés d'assurer la gestion de flux des signaux. Grâce à eux, une régulation est effectuée entre les blocs de calcul qui peuvent être temporairement incapables de s'exécuter à cause par exemple d'une indisponibilité au niveau des données d'entrée, ou encore de l'incapacité du reste du réseau à stocker les résultats s'ils étaient produits.

Le second problème vient de la capacité finie des ressources matérielles et tampons bornés aux interconnections : c'est à dire les lignes de stations de relais. Depuis leur invention les stations de relais ont été un sujet d'attention pour un nombre de groupes de recherche. Des modélisation extensives, caractérisations et

analyses ont été fournies dans [23, 32, 30]. Jusqu'à présent le niveau de modélisation n'avait pas atteint un niveau de formalisation complet, ainsi les preuves de correction étaient encore informelles, ou encore basées sur des allusions ou encore de simulations de modèles d'exécution. Nous allons utiliser un papier écrit par Casu et Macchiarulo [27], qui fournit un excellent point de départ pour notre modélisation.

Chaque station de relais peut être conçue comme une cellule faisant partie d'une ligne de n stations de relais, décomposant ainsi un fil en $n+1$ sections correspondantes à $n+1$ cycles d'horloge. Les stations de relais implantent un protocole spécifique, qui sera dans un sens préservé par chaînage. Il ne fera qu'augmenter le nombre de latences "obligatoires". Chaque station de relais peut recevoir un signal "donnée valide" de son prédécesseur (soit un shell autour d'une IP ou une autre station de relais), et passera ce signal à son successeur *lors du prochain cycle d'horloge*. La station de relais peut aussi recevoir dans la direction inverse un signal de *"régulation"* implantant une fonction de *"contre-pression"* (back-pressure) pour indiquer que le nœud successeur est incapable d'accepter plus de données. Dans ce cas la station de relais doit s'abstenir d'envoyer sa valeur (donnée) et doit la conserver sur place. Elle doit aussi être capable de recevoir la nouvelle donnée dans ce cycle puisque le nœud précédent n'a pu être averti de la congestion ; et si nécessaire elle propagera le signal de congestion au nœud précédent *lors du prochain cycle d'horloge*. Les délais du *prochain cycle* sont nécessaires afin de respecter l'hypothèse physique effectuée sur la latence. Bien sûr il y a aussi le cas où aucune donnée valide n'est transmise depuis le nœud précédent car en amont des unités de calculs sont arrêtées temporairement à cause d'un manque sur au moins une de leurs entrées. Il doit être aussi noté que n'importe quelle station de relais nécessite la capacité de conserver *deux* valeurs simultanément, dans le cas où nous ne pouvons propager la valeur courante alors que cette dernière en reçoit une autre simultanément. Elle peut aussi être vide, si des données valides sont produites plus lentement que consommées.

En fait le rôle des stations de relais est double : elles implantent l'ordonnancement dynamique nécessaire afin de gérer correctement les risques de congestion à l'aide du mécanisme de "contre-pression" (back-pressure) ; elles fournissent aussi l'espace de stockage temporaire pour les données tant qu'elles ne peuvent être envoyées plus loin dans la ligne de stations de relais. Ce second rôle est discutable : si la donnée était autorisée à continuer sa route, elles auraient pu être stockées directement au niveau du shell destinataire, si ce dernier fournissait la même capacité de stockage que la ligne de stations de relais. C'est à dire que, nous aurions pu déplacer tout le stockage sur un seul emplacement spatial (en conservant des répéteurs) et simplifier les différentes itérations nécessaires dans le cadre de la synthèse physique : cela fut noté dans [27] (mais n'est pas toujours faisable). La régulation du trafic et le mécanisme de contre-pression sont toujours

obligatoires, autrement il y aurait un risque de dépassement de capacité.

Le mécanisme de back-pressure montre l'effet de la rétro-propagation d'information sur la congestion et l'"*embouteillage*" arrivé en *aval*. Il le fait seulement lorsque c'est nécessaire, mais *lorsque c'est faisable au plus tôt*, tout en respectant les latences nécessaires pour voyager au travers de ces "longs" fils.

3.1 Notre modélisation formelle de la conception LID

Dans cette section, nous revisitons la modélisation formelle des *stations de relais* (relay stations) et des *shells*, qui sont des éléments de connexion spécifiques utilisés dans la théorie des systèmes dits insensibles à la latence (LID). Les stations de relais sont chargées d'effectuer la régulation du trafic des signaux/données pour éviter famine, interblocage et congestion des blocs (IP) synchrones locaux ; tout en respectant les contraintes physiques c'est à dire dans ce cadre les latences. Depuis la proposition de Carloni *et al*, la structure et les comportements de ces stations de relais et des shells ont été amplement caractérisés et analysés. Mais les travaux précédents n'ont pas fourni une description complètement formelle et suffisamment précise pour être amenable au stade de la vérification formelle ; à la place, principalement des modèles de simulation ont été développés. À cause de la nécessité de précision de l'ensemble, nous pensons qu'une telle description formelle est nécessaire. Nous décrivons une telle tentative dans cette section.

Cette section est organisée de la manière suivante :

Dans la sous-section 3.1.1 nous proposons des contraintes formelles et des besoins abstraits qui doivent être satisfaits par les modèles des stations de relais. Nous partons du modèle de [27], qui en lui-même résume les travaux précédents. Nous proposons notre modèle formel sous la forme d'un syncchart qui dispose de caractéristiques régulières et des spécifications temporelles pour les signaux de sortie. Ce modèle est amenable à une description en Esterel [10] ou SyncCharts [4, 5], permettant ainsi d'utiliser des méthodes formelles et notamment des techniques de model-checking [12]. Bien sûr il est aussi possible de fournir une traduction directe vers des langages tels que `VHDL, Verilog` par exemple, mais nous gagnons en flexibilité permettant de décrire facilement la combinaison de différentes stations de relais en des fils de plus grande latence par exemple. Dans la sous-section 3.1.2 nous spécifions formellement un certain nombre de propriétés de correction qui peuvent être établies sur une ligne de stations de relais. Bien sûr le model-checking ne permet pas de raisonner sur des modèles paramétriques (où ici le paramètre en l'occurence serait la longueur en latence n de la ligne), ainsi nous devons instancier pour différentes constantes de longueur. Nous décrivons ensuite le shell (ici très proche de la version de [27]) dans la sous-section 3.1.3. De la même manière nous effectuons une vérification à l'aide d'un model-checker

sur des propriétés de correction.

3.1.1 Station de Relais

Nous pouvons maintenant décrire la station de relais. Le but est d'implanter des canaux de communication de taille bornée qui divise les longs fils en sections, de telle manière que les signaux/données seront propagés d'une section à une autre seulement lors du prochain coup d'horloge. Similairement les signaux nécessaires pour implanter le contrôle de congestion (contre-pression) doivent aussi respecter ces délais de voyage. Dans ce but, les *stations de relais* ont été introduites dans [62]. Ce sont des éléments matériels spécifiques qui fournissent l'interface de connexion entre les sections (et aussi les shells aux extrêmités des canaux). Ces éléments doivent pouvoir stocker les données *"en route"* bien sûr (fonction de *wire-pipelining*) mais aussi pour recevoir les données additionnelles qui peuvent arriver à cause de congestions, le canal en aval ne pouvant les accepter.

Modélisation de la Station de Relais

Malgré le nombre de publications décrivant les stations de relais dans la littérature, elles sont généralement dépeintes de manière informelle. La plupart du temps, les contraintes précises explicitant les temps physiques nécessaire (en cycles d'horloge), ni leur modèle formel et leur correction sont complètement décrits. Le papier parvenant le plus prêt de ces buts est [27]. Cependant les auteurs n'utilisent pas une modélisation purement synchrone dans leur machine à états finis.

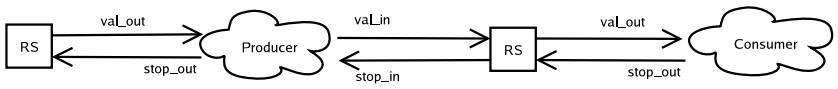


FIG. 3.1 – Station de Relais - Diagramme Bloc

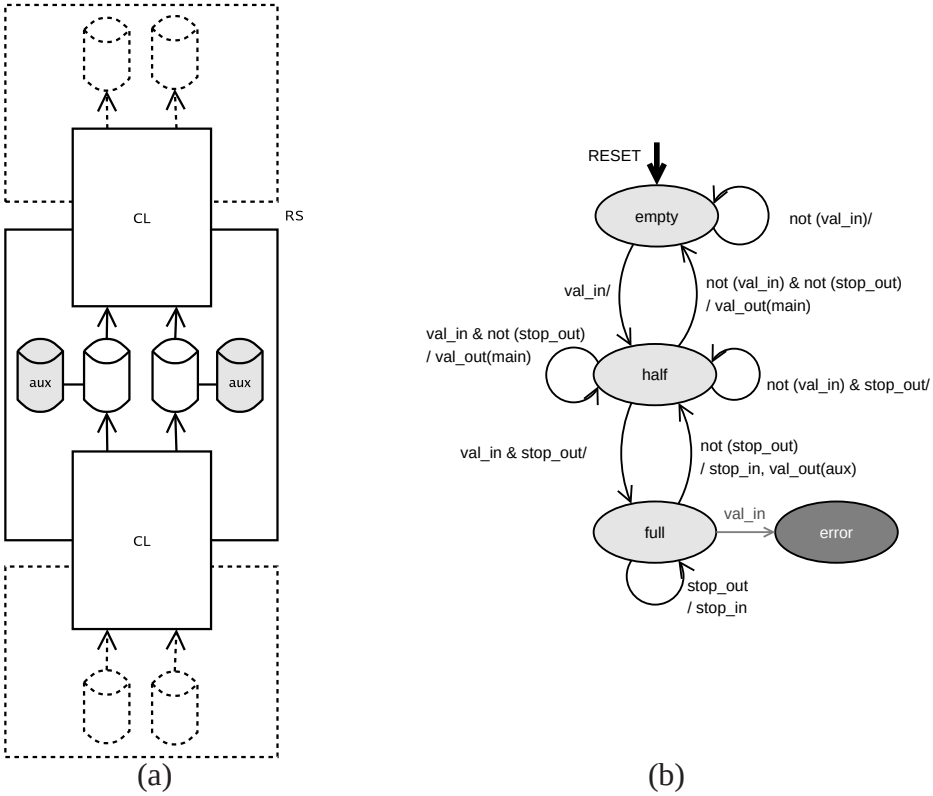
Nous reprenons de [27] l'interface des signaux d'entrées/sorties (malgré un nommage de signaux peu intuitif). Cette interface est illustrée dans la figure 3.1. La donnée d'entrée est représentée par le signal *val_in* qui est vrai (il correspond à un $\neg\tau$, notation utilisée dans les articles antérieurs sur LID, c'est à dire que la valeur de la donnée est significative). C'est un signal booléen pur (nous pouvons abstraire les valeurs des données). La station de relais cède la donnée avec le signal *val_out* correspondant. Concernant la contre-pression, la station de relais peut recevoir un ordre d'arrêt avec le signal *stop_out*. Elle le transmet alors en amont avec un signal *stop_in* (ainsi *stop_out* qui est une entrée et *stop_in* une sortie) lorsqu'il y a congestion.

Contraintes physiques : Il est important de noter que les signaux/données ne peuvent être propagés combinatoirement d'une section à une autre :

– $val\_in \hookrightarrow_{next} val\_out$.

– $stop\_out \hookrightarrow_{next} stop\_in$.

D'un autre côté, il peut y avoir des relations combinatoires entre *stop_in* et *val_in* (respectivement entre *stop_out* et *val_out*, comme ils sont dans la même section).



(a) (b)

FIG. 3.2 – (a) Structure de la station de relais (b) syncchart de la station de relais

Ainsi les stations de relais ont besoin de registres (FF par exemple) pour retenir les signaux entre les réceptions et les propagations. En fait, comme montré dans [62], elles ont besoin de deux emplacements, dans le cas où une nouvelle donnée arrive pendant que la courante ne peut être propagée. Après, le mécanisme de congestion est supposé garantir qu'aucune autre donnée puisse être reçue (et ainsi perdue) car elles sont retenues quelque part en amont. Nous obtenons la figure abstraite 3.2 (a).

Station de Relais - syncchart Nous représentons dans la figure 3.2 (b) la station de relais comme un syncchart [4, 5], avec des états explicites, gérant ainsi à la

fois les sorties et les fonctions de prochain état et une implantation possible de ce syncchart avec les chemins de données dans la figure 3.3. Maintenant, nous décrivons ce syncchart utilisant pour encodage des états le nombre de registres libres à l'intérieur de la station de relais.

La SSM (Safe State Machine - appellation commerciale de SyncChart) contient 3 états qui correspondent à l'occupation des registres :

`empty` *lorsqu'aucune donnée n'est présente dans la station de relais* ; dans cet état la station de relais attend une donnée valide en entrée, et l'entrepose dans son registre `principal` (passe à l'état `half`). Les signaux *stop_out* sont ignorés, et non propagés en amont puisque la cellule est capable d'absorber le trafic.

`half` *lorsqu'il y a une donnée* ; Alors la station de relais transmet la donnée courante (le signal reçu précédemment) si elle n'a pas reçu de signal d'arrêt *stop_out* dans ce cas (souvenez-vous que cette relation combinatoire est correcte, lorsque nous sommes à l'intérieur d'une section). S'il y a arrêt, la station de relais conserve la donnée, mais doit aussi accepter la nouvelle donnée potentielle arrivant en amont (parce qu'elle n'a pas encore envoyé un signal de contre-pression). Dans le second cas elle devient pleine, avec la seconde valeur occupant le registre d'*"urgence"* c'est à dire le registre `aux`illiaire. Si la station de relais peut transmettre (*stop_out* false), elle retourne alors sur l'état `empty` ou reprend une nouvelle donnée valide, restant alors dans le même état. Elle n'aura pas besoin de propager la contre-pression (dans le prochain cycle), car elle dispose de ressources de stockage suffisantes.

`full` *lorsqu'elle contient deux données* ; alors dans n'importe quel cas le signal *stop_in* sera envoyé, propageant à la section en amont le signal *stop_out* reçu le coup d'horloge précédent. Si la station de relais ne reçoit pas elle-même un nouveau *stop_out*, alors la ligne en aval dispose de suffisamment de place pour que nous transmettions la donnée ; autrement elle la garde et reste en arrêt.

`error` est un état qui ne doit jamais être atteint (à la manière d'une condition *assume/guarantee*). L'idée est qu'il s'agit d'une précondition affirmant que l'environnement n'enverra jamais de signal *val_in* lorsque la station de relais émettra le signal *stop_in*. Ceci doit être étendu à n'importe quelle combinaison de stations de relais, et construit une condition sur les entrées du système. La propriété est préservée en tant que postcondition, comme chaque station de relais garantira que le signal *val_out* n'est jamais émis lorsqu'un *stop_out* arrive.

NB : Un signal est émis (noté par */*) seulement lorsque la garde est satisfaite.

La figure 3.3 a) présente une implantation possible du syncchart de la figure 3.2. La figure 3.3 b) illustre le chemin de données de la station relais.

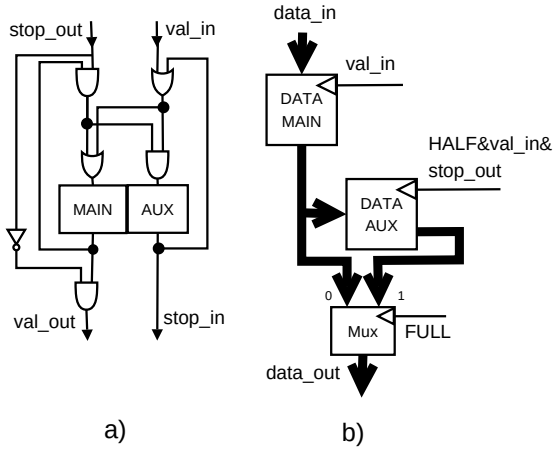


FIG. 3.3 – Station de Relais - a) Logique de Contrôle b) Chemin des Données

3.1.2 Propriétés de correction de la Station de Relais

Dérivant des remarques de la discussion précédente, nous pouvons décrire un certain nombre de propriétés de correction sur la station de relais, ou une ligne de stations de relais (ou plus tard, un réseau comprenant des shells et perles). Rappelons que les critères de correction pour la vivacité (vue comme l'absence de blocage et l'absence de congestion) ont été déja établis comme marquage de graphe de réseau de Petri, lié à l'initialisation de ce dernier. Quelques exemples de propriétés additionelles sont :

- les stations de relais ne peuvent pas avoir de dépassement de capacité.
- l'ordre des données est préservé.
- à n'importe quel point du temps, le nombre de données valides produites d'une ligne est borné relativement au nombre entré :

  $$\#(val\_in) + Init\_line \leq \#(val\_out) \leq \#(val\_in) + Init\_line + 2 \times length\_line$$

  où $Init\_line$ est le nombre de données initialement résidentes dans les lignes des stations de relais, et $length\_line$ est le nombre de stations de relais.
- une ligne de n stations de relais ne peut signaler la congestion à sa source à moins qu'elle ne reçoive suffisament de signaux de contre-pression, étant donné le contexte initial.

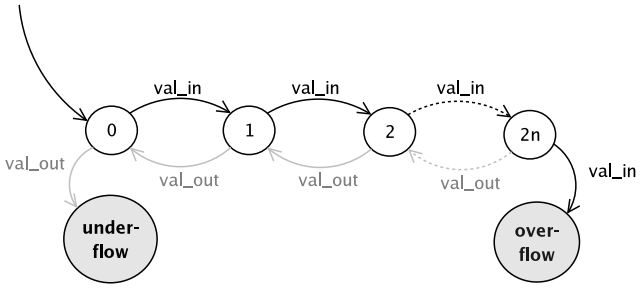


FIG. 3.4 – Observateur de dépassement pour station de relais

– inversement, une ligne recevant suffisamment de contre-pression sera remplie et enverra un signal de congestion.

La première propriété peut être vérifiée par l'observateur de la figure 3.4.

La vérification va ainsi consister à prouver que ces états ne sont pas atteignables dans toutes les stations de relais. La seconde propriété peut être modélisée dans un cas restreint en *"marquant"* les signaux de données successifs avec des indices, et ensuite vérifier que ces indices sont retournés par la ligne dans le même ordre que lors de l'entrée dans l'autre extrêmité. Un schéma simple est d'alterner les marques 0 et 1 fournissant une vérification du type *protocole du bit alterné*.

Nous vérifions ces propriétés par model-checking, avec (un petit intervalle) des constantes remplaçant les paramètres entiers, et observateurs construits depuis ces formules.

Nous décrivons maintenant d'autres propriétés qu'une station de relais doit satisfaire, ainsi que des *liens* qui sont une suite de stations de relais que nous noterons $L_n(k)$ de n stations de relais successives et contenant k valeurs (souvenez-vous qu'une ligne de n stations de relais peut stocker $2n$ valeurs).

Sur une station de relais unique :

– $\Box\ \neg(stop\_out \wedge val\_out)$ (le contrôle de flot agit immédiatement) ;
– $\Box\ ((stop\_out \wedge X(stop\_out)) \Rightarrow X(stop\_in)$ (une *RS* bloquée est remplie en deux étapes)

où $\Box$, $\Diamond$, $\mathcal{U}$ et X sont les opérateurs classiques de logique temporelle *Always*, *Eventually*, *Until* et *Next*.

Des propriétés plus intéressantes peuvent être vérifiées sur des lignes de stations de relais (nous supposons le renommage des signaux $stop\_\{in,out\}$ et $val\_\{in,out\}$ forment les interfaces d'entrée/sortie de la ligne globale $L_n(k)$) :

– $\Box\ (\neg stop\_out \Rightarrow \neg X^n(stop\_in))$ (les places libres se propagent en arrière) ;
– $\Box\big((stop\_out\ \mathcal{U}\ X^{(2n-k)}(true)) \Rightarrow X^{(2n-k)}(stop\_in)\big)$ (dépassement) ;
– $\big(\Diamond\ val\_in \wedge \Box(\Diamond(\neg stop\_out)\big) \Rightarrow \Diamond val\_out$ (si le trafic n'est pas complète-

ment bloqué au-delà d'un point donné, alors les jetons passeront au travers) La première propriété est vraie pour n'importe quelle ligne de taille *n*, la seconde de n'importe quelle ligne contenant initialement au moins *k* jetons, la troisième de n'importe quelle ligne.

Nous avons implanté les stations de relais et des lignes de stations de relais dans le langage synchrone `Esterel` et model-checké des combinaisons de ces propriétés en utilisant *EsterelStudio*. [1]

3.1.3 Shell

Le but du *Shell* est d'activer l'IP exactement lorsque toutes les données sont disponibles pour chaque entrée, et qu'il y a suffisamment de place disponible pour stocker les résultats sur les sorties. Cela correspond à la notion de *clock gating* dans les circuits :

Le *Shell* fournit une horloge logique qui active l'IP. Bien sûr, il est nécessaire que le composant soit physiquement capable de fonctionner avec de telles horloges irrégulières (une propriété dénommée *patience* dans le vocabulaire du *LID*), mais cet aspect technologique est transparent au niveau de modélisation dans lequel nous nous plaçons. Aussi, il doit être rappelé que l'IP est supposée produire une donnée sur toutes ses sorties alors qu'elle en consomme une sur chacune de ses entrées à chaque étape de calcul. Ceci n'implique pas un comportement strictement combinatoire, car les IPs peuvent contenir des registres.

L'interface d'un *Shell* consiste en des signaux `val_in` et `stop_in` indexés par le nombre des entrées à ce *Shell*, et des signaux `val_out` et `stop_out` indexés par le nombre des sorties. Il y a un signal de sortie *clock* pour forcer l'exécution du composant local. Ce signal et tous les signaux de sorties sont synchrones.

Le comportement opérationel du *Shell* est montré comme un circuit synchrone dans la figure 3.5 (a), où chaque module d'entrée `i` doit être instancié avec la figure 3.5 (b), avec les signaux proprement renommés, qui finalement contrôlent le chemin de donnée de la figure 3.5 (c). Le *Shell* est combinatoire, cela prend un cycle d'horloge pour passer des stations de relais en amont du *Shell*, passer au travers de ce dernier, et finalement arriver aux stations de relais en sortie du *Shell*. La *Perle* (IP) est *Patiente*, l'état de la *Perle* est seulement changé lorsque l'horloge (périodique ou sporadique) arrive.

Modélisation du Shell

Ici notre modèle suit d'assez près celui créé par Casu et Macchiarulo [27]. Il est illustré dans la figure 3.5.

[1] *EsterelStudio*™ est une marque d' *Esterel Technologies*

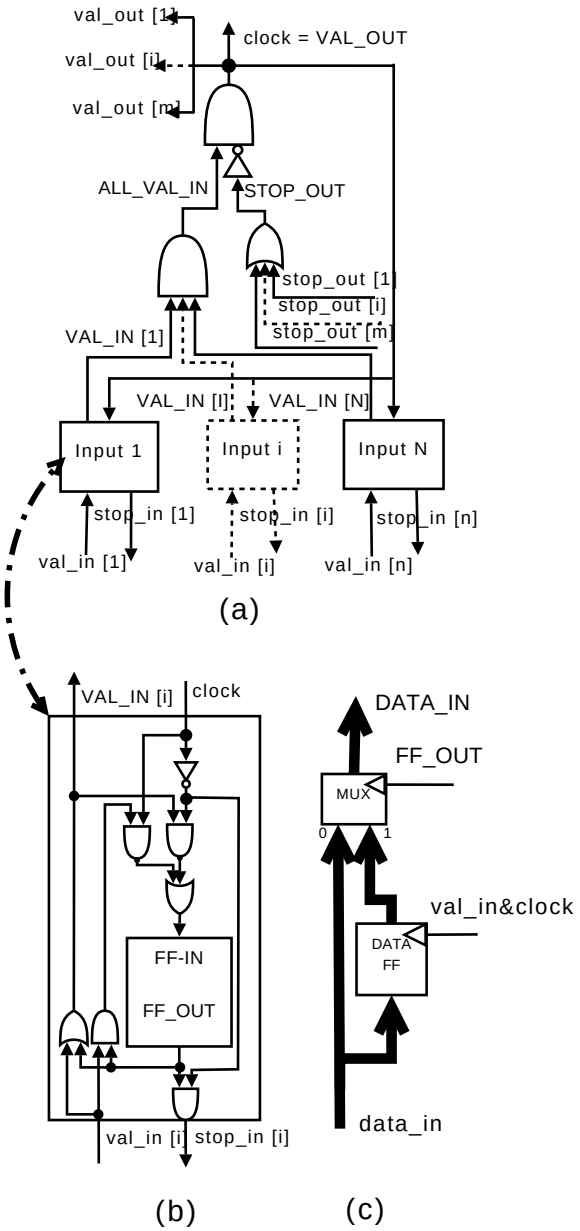


FIG. 3.5 – Circuit du Shell

Équations du Shell

- $stop\_in_i = (\bigvee stop\_out_{0...N} \bigvee \neg VAL\_IN_{0..N_3}) \bigwedge flipflop_{out}$.
- $VAL\_IN_i = val\_in_i \bigvee flipflop_{out}$.
- $val\_out_{0...N_1} = clock = \neg \bigvee stop\_out_{0...N_2} \bigwedge VAL\_IN_{0...N_3}$.

– $flipflop_{in} = (VAL\_IN_i \bigwedge stop\_out_{0...N_2}) \bigvee$
$(VAL\_IN_i \bigwedge \neg VAL\_IN_{0...N_3}) \bigvee$
$(val\_in_i \bigwedge flipflop_{out} \bigwedge clock)$.

Comme mentionné précedemment, nous pouvons considérer le cas où les shells et les perles peuvent ne pas avoir de délais de propagation (tant qu'il n'y a pas de boucle combinatoire entre les shells, c'est à dire qu'il existe au moins une station de relais).

Les shells auront besoin d'avoir la possibilité d'entreposer des données qui sont déjà arrivées, attendant les autres données manquantes.

Le Shell fonctionne de la manière suivante :

– L'horloge (*clock*) de la perle interne et tous les signaux de sorties val_out_i sont générés seulement lorsque nous avons tous les *val_in*, tout en ayant *stop* qui est faux. Le signal de *stop* interne par lui-même représente la disjonction de tous les signaux $stop\_out_j$ entrants depuis les canaux sortants.
– le registre tampon d'un canal d'entrée est utilisé tant que toutes les autres données d'entrée ne sont pas disponibles.
– ainsi, l'horloge interne (*clock*) est fausse lorsque un $stop\_out_j$ de contre-pression est mis à vrai, ou un val_in_i entrant est faux. Dans un tel cas les registres déjà occupés conservent leur *vraie* valeur, pendant que les autres peuvent recevoir une donnée valide *"juste à l'instant"*.
– les signaux $stop\_in_i$ sont envoyés à tous les canaux dont les registres correspondants étaient déjà chargés (une donnée a été reçue auparavant, et n'a pas encore été consommée), pour prévenir les autres de ne pas propager de valeur dans le cycle d'horloge. Bien sûr un tel signal ne doit pas être émis dans le cas où la donnée est en train d'être envoyée, autrement il y aurait un problème de causalité (et un cycle combinatoire).
– Les registres sont remis à zéro lorsque l'horloge de la perle est activée, c'est à dire que l'on consomme les données. Le signal $stop\_in_i$ est levé seulement lorsque la $flipflop_i$ avait déjà une valeur et que le signal *stop_out* est présent .

Nous devons nous rappeler la contrainte demandée par les stations de relais pour avoir un fonctionnement correct, c'est à dire que sur chaque canal de sortie depuis le producteur (cas du shell), ainsi $stop\_out_j \Rightarrow \neg val\_out_j$, est vrai ici.

3.1.4 Propriétés de correction du Shell

De la même manière nous avons effectué des expérimentations sur les *Shells* en employant la technique de model-checking en utilisant ESTEREL STUDIO :

– $\square\big((\exists j, stop\_out_j) \vee \Rightarrow \neg clock\big)$ où j est l'index d'une entrée ;

– $\Box\big((\exists j, stop\_out_j) \Rightarrow (\forall i, \neg val\_out_i)\big)$ où j/i est un index d'entrée/sortie respectivement ;
– $\Box\big((\forall j, \neg stop\_out_j \wedge \neg X(stop\_out_j)) \Rightarrow (X(clock) \Rightarrow \exists i, X(val\_in_i))\big)$ où j, i sont les indexes d'entrée (si le shell n'a pas été suspendu à un instant par une congestion en sortie, et il active sa perle le prochain instant, alors il doit le faire car il reçoit une nouvelle valeur sur une entrée au prochain instant).

D'un autre côté, la plupart des propriétés usuelles ici nécessitent du *sucre syntaxique* de façon à être plus facilement formulées (tel que "un jeton doit arriver sur chaque entrée avant ou en même temps que le shell active sa perle, mais ils peuvent arriver dans n'importe quel ordre"). Comme dans le cas des stations de relais, la correction dépend d'une hypothèse effectuée sur l'environnement qui est $\forall i, stop\_in_i \Rightarrow \neg val\_in_i$, signifiant que des composants en amont ne doivent pas envoyer de valeur lorsqu'une partie du système est bloquée.

Gardant en mémoire les stations de relais, nous voulons montrer cette propriété :

– la donnée ne peut être acceptée avant que la précédente soit consommée : l'ordre des données est préservé.

Cette propriété peut être vérifiée facilement car le shell est connecté *combinatoirement* à la station de relais (ou un autre shell) et ainsi la station de relais ne peut envoyer n'importe quelle donnée au shell lorsque ce dernier a déjà une donnée présente. Le shell peut avoir seulement une valeur dans chacun de ses canaux comme dit auparavant il ne peut écrire par dessus ou perdre cette donnée avant que toutes les données nécessaires soient présentes pour réagir. L'ordre des données est préservé, car par hypothèse le réseau d'interconnection est seulement point à point, et ne peut perdre ou altérer l'ordre des données. Le shell attend toutes les données et réagit, ainsi l'ordre partiel du design désynchronisé est compatible avec celui de la spécification synchrone. Nous pouvons aussi appliquer la vérification avec le bit alterné sur ce cas. Le shell ne bloque pas car nous avons établi comme une condition de marquage de Marked Event Graphs.

Le modèle a été construit avec EsterelV5/EsterelStudio et formellement vérifié avec le model-checker Xeve [12]/Prover et les outils de visualisation graphique Autograph[69]. Xeve est un model-checker basé sur les BDDs qui peuvent en plus de montrer la propriété, fournir un automate minimal canonique pour le composant (jusqu'à la bisimulation). Autograph permet à l'utilisateur d'afficher graphiquement l'automate minimal (et reconnaître ses caractéristiques, lorsque le nombre d'états reste gérable).

3.1.5 Résumé

Dans cette section nous avons présenté une implémentation synchrone de la théorie des systèmes insensibles à la latence vérifiée formellement à l'aide d'un

model-checker.

Nous avons décrit, implémenté et vérifié formellement une *station de relais* et un *shell*.

La station de relais est un répéteur de signal intelligent implantant une partie du protocole de “back-pressure” permettant d’assurer la correction du comportement du système insensible à la latence par rapport à la spécification synchrone. L’implantation utilise un simple automate disposant de 3 états associé aux différents statuts des 2 registres nécessaires pour le stockage des données.

Le shell est une interface permettant d’assurer conjointement avec les stations de relais la “synchronisation” des données ainsi que le clock-gating nécessaire par rapport à la spécification synchrone. L’implantation du shell est combinatoire fonction des statuts des entrées et sorties des stations de relais et utilise des FIFOs “court-circuitables”.

Un ensemble de propriétés à vérifier formellement sont détaillées, à la fois pour chaque composant *shell* et *station de relais*, des lignes de stations de relais, et la composition station de relais et shell.

Le contenu de ce chapitre a donné lieu à une publication dans la conférence FMGALS en 2005 [14]. Ce travail a été effectué conjointement avec Jean-Vivien Millo et Robert de Simone.

Chapitre 4

Ordonnancement statique et régimes stationnaires k-périodiques

On gagne une bataille en connaissant le rythme de l'ennemi, et en utilisant un rythme auquel il ne s'attendait pas - Miyamoto Musashi

La théorie des systèmes dits insensibles à la latence a été récemment inventée afin de faire face au problème de *Timing Closure* dans les circuits et programmes synchrones. L'idée est d'autoriser l'introduction de latences (entières) fixes arbitraires sur les fils ou d'autres médias de communication permettant de faire transiter des signaux/données. Après des mécanismes tels que les *Shells* et les *stations de relais* sont introduits afin d'implanter le protocole de contrôle de flot (contre-pression/back-pressure) nécessaire, ainsi toutes les données ayant des temps de traversée plus courts peuvent attendre les données en retard avant que toutes ces données soient consommées simultanément par l'élément de calcul concerné. Ces mécanismes peuvent être représentés comme des composants synchrones dans un environnement global inspiré du monde asynchrone. Malgré leur efficacité, les stations de relais et les mécanismes de contre-pression ajoutent de la complexité au système dont le comportement est ultimement répétitif. Entre autres, ce sont les boucles les plus *"lentes"* qui régulent le trafic et l'organisent à leur rythme. Cet ordonnancement répétitif spécifique a été étudié en détail, sous le nom de "Problème Central Répétitif" [19], et des résultats ont été établis prouvant que les solutions périodiques dénommées *k-périodiques* sont optimales et peuvent toujours être atteintes. Mais l'implantation utilisant des circuits synchrones usuels dans le contexte des systèmes dits-insensibles à la latence n'a jamais été effectuée.

Nous allons traiter ces problèmes dans ce chapitre, en utilisant une représentation explicite des ordonnancements en tant que mots périodiques sur $\{0,1\}^\star$ issus

de la théorie récente des systèmes *N-Synchrones* [33].

Contributions L'objectif principal est de fournir une version ordonnancée statiquement du réseau de calcul synchrone avec des latences données. Nous pouvons penser à ajouter plus de latences que celles déjà imposées dans le but de ralentir le débit et d'arriver à réguler exactement le trafic. Les nouvelles latences sont en quelque sorte "virtuelles", comme elles ne sont pas définies de manière unique et peuvent être utilisées pour un *re-design* des éléments de calcul (minimisation de la surface et de la puissance). Le but est d'*égaliser* toutes les "longueurs" des chemins (comptées en cycles d'horloge) tout en conservant la même règle d'évaluation synchrone relaxée, de telle manière à ce que tous les mécanismes de contrôle de flot, de contre-pression soient inutiles et que les stations de relais soient alors simplifiées en de simples registres. Mais malheureusement cela n'est pas toujours possible, à cause du fait que la solution exacte nécessite dans le cas général un délais rationnel (plutôt qu'un entier) à être inséré.

Notre but sera d'insérer dans un premier temps autant de latences entières que possible, et après seulement de traiter les parties fractionnaires restantes qui empêchent d'avoir une égalisation parfaite. Ces parties fractionnaires seront modélisées en utilisant un élément particulier que nous appelons *registre fractionnaire*. Des tentatives antérieures ont été effectuées dans [28], mais les auteurs avaient besoin de diviser les cycles d'horloge en plus petites phases, et nous ne souhaitons pas cela. Nous avons besoin d'introduire un élément qui *"capture"* des jetons de temps en temps. Et nous avons besoin de préciser explicitement lorsque nous "enlevons" ces jetons.

Pour représenter explicitement les ordonnancements des activités sur les nœuds de calculs nous utilisons la notation de la théorie des processus *N-synchrones* [33], où une telle notion a été introduite précédemment. Nous identifions un certain nombre de phénomènes apparaissant lorsque des boucles avec des débits différents sont présents. Nous parlerons aussi des problèmes de la phase d'initialisation et de la reconnaissance de régimes périodiques.

4.1 État de l'Art

4.1.1 Rappel

Les designs d'aujourd'hui effectués dans le cadre de systèmes complexes sur puce ont de gros problèmes de *Timing Closure* et de *synchronisation d'horloges*. La théorie des systèmes dits-insensibles à la latence [24, 63, 62, 21] a été introduite afin de résoudre ce problème de Timing Closure. Elle le fait en définissant des éléments de stockage qui peuvent s'installer le long des fils, divisant ces der-

niers d'une latence donnée en différentes sections. Un schéma d'ordonnancement dynamique est alors employé afin d'éviter les congestions des données, cette implantation est effectuée en utilisant des signaux additionnels. En fait les routes les plus rapides sont ralenties, régulées au rythme des plus lentes.

D'un autre côté, la théorie générale des Marked Event Graphs annotés par des poids nous apprend qu'il existe un ordonnancement répétitif statique pour de tels systèmes [19, 8]. De tels ordonnancements statiques *k*-periodiques ont été appliqués à des problèmes de software pipelining [37, 66] (où nous cherchons à trouver plus de parallélisme en dépliant partiellement des boucles afin de saturer des unités de calculs sur un processeur VLIW, par exemple), et plus tard sur les designs LID avec [28]. Mais ces solutions font généralement peu attention au type des éléments de stockage conservant les valeurs dans le système ordonnancé, et leur adéquation vis à vis d'une représentation circuit matérielle. Par exemple, dans [28] l'horloge de base est divisée en plusieurs phases, une solution que nous ne trouvons pas adéquate ici (pour l'aspect théorique et non technique : nous nous interdisons l'utilisation d'une horloge rationelle). Entre autres, dans ces solutions l'allocation précise des cycles d'horloges aux nœuds de calculs n'est pas explicite. Exprimer un ordonnancement statique précis qui utilise les éléments synchrones est désirable pour un certain nombre de problèmes. Il peut être facilement synthétisé bien sûr, mais nous pouvons aussi évaluer sa consommation électrique, en essayant d'alterner du mieux que nous pouvons les phases de calculs avec les phases de transport. Il peut aussi être utilisé comme une base à l'introduction des *modes* de contrôle qui sont couramment absents du modèle (qui effectue toujours les mêmes calculs dans un ordre partiel). Nous ne considèrerons pas ce problème dans ce chapitre.

Structure du chapitre : Dans un précédent chapitre nous avons introduit en détail le *contexte de modélisation* des réseaux de calculs et des variations des sémantiques attachées aux règles d'évaluation. Nous avons décrit les sémantiques classiques synchrone et asynchrone. L'introduction des poids (des latences ici) crée un modèle intermédiaire entre ces deux sémantiques.

Dans la section suivante, nous introduisons la théorie *N*-synchrone. Cette notation nous permet d'exprimer le problème général de l'égalisation des latences que nous souhaitons traiter. Nous finirons cette section par un résumé de résultats connus importants sur les ordonnancements *k*-périodiques pour les Marked Event Graphs annotés par des poids.

Ensuite nous décrivons notre approche et les différentes étapes de l'algorithme permettant d'introduire des latences entières, calculer les ordonnancements après une phase d'initialisation et un régime périodique ; d'identifier les endroits où nous pouvons rajouter ces latences virtuelles entre les différentes routes ayant des

débits différents. Le but est de maintenir le débit originel du système, c'est à dire celui de la boucle la plus lente, qui est le débit maximum atteignable par le système dans tous les cas. Une description formelle du *registre fractionnel* est fournie, avec ses conditions d'application. Les problèmes d'une implantation efficace sont aussi mentionnés (nous avons construit un prototype implantant ces idées). Nous proposons un certain nombre d'exemples afin de montrer les difficultés et nous finirons en considérant d'autres thèmes.

4.1.2 N-Synchrone

La théorie N-Synchrone s'inspire du modèle synchrone. Mais, plutôt que de considérer des sous-horloges engendrées par des signaux, elle considère des patrons de sous-échantillonnage (ultimement k)-périodiques, fonctionnant aussi avec les opérateurs *when* et *merge*. Ce modèle fut introduit par Cohen *et al.* dans [33]. Ce modèle permet de capturer et d'étendre la notion d'exécution périodique du synchrone en utilisant comme syntaxe effective des mots binaires infinis se conformant à la grammaire suivante :

- $word = init$ **(** $period$ **) .**
- $init = [\ \mathbf{0} \mid \mathbf{1}\ ]^{*}$**.**
- $period = [\ \mathbf{0} \mid \mathbf{1}\ ]^{+}$**.**

où *init* permet de décrire la phase que nous qualifierons d'initiale ou transitoire et un mot $(period)$ qui est une répétition périodique infinie, une occurence de 1 signifie que l'horloge est *présente* (et respectivement 0 lorsqu'elle est *absente*).

Définition 4 (Longueur)**.** La longueur du mot *period* est notée $|period|$.

Définition 5 (Nombre d'occurences de x)**.** Les nombres d'occurences de 1 et 0 sont notés respectivement par $|period|_1$ et $|period|_0$.

Définition 6 (Index)**.** Nous appelons index d'un mot la k-ième occurence de 1 de la partie périodique du mot que nous noterons : $[period]_k \in \mathbb{N}$, avec $k \in [1, |period|_1]$.

Définition 7 (Ordre)**.** Soit la relation d'ordre $\preceq$ qui est un ordre partiel $word_a \preceq word_b \iff \forall n \geq 1, word_a[n]_1 \leq word_b[n]_1$.

Cette relation abstrait la relation de causalité qui vérifie que toutes les sorties sont produites avant que les consommateurs les nécessite. Par exemple $01 \preceq 0001$.

Cependant nous ne pouvons rien dire entre $\alpha = 0110$ et $\beta = 1001$ pour cette relation car $[\alpha]_1 = 2 > [\beta]_1 = 1$ et $[\alpha]_2 = 3 < [\beta]_2 = 4$.

Définition 8 (Max,Min)**.** Deux autres opérateurs sont introduits : la borne haute et basse de deux mots infinis. Ces opérateurs sont notés $\sqcup$ et $\sqcap$ respectivement et définis de la manière suivante :

– $\forall n \geq 1, [a \sqcup b]_n = Max([a]_n, [b]_n)$.
– $\forall n \geq 1, [a \sqcap b]_n = Min([a]_n, [b]_n)$.

$\sqcup$ est utile car il peut être vu comme un ordonnancement avec dates au plus tôt entre deux entités, le calcul peut commencer lorsque toutes les entrées sont prêtes dans notre modèle précédent. *NB :* Nous les noterons aussi par *Max* et *Min* lorsqu'il n'y pas d'ambigüité.

Maintenant nous pouvons combiner $\alpha \sqcup \beta = 0101$ et obtenir un ordonnancement au plus tôt.

Propriété 1. *Max ($\sqcup$) et Min ($\sqcap$) sont monotones.*

Propriété 2. *Max ($\sqcup$) et Min ($\sqcap$) vérifient la relation d'ordre $\preceq$ et $\succeq$ respectivement.*

Définition 9 (Early,Late)**.** Nous définissons deux générateurs simples de mots que nous appelerons *one* et *zero* de la manière suivante :

– $one_i = 1^i$.
– $zero_i = 0^i$.

puis nous introduisons deux autres générateurs que nous dénommons $early$ et $late$ définis par :

– $early(i, j) = one_i.zero_{j-i}$,
– $late(i, j) = zero_{j-i}.one_i$

avec $i \leq j$ où i est le nombre d'occurences de 1 et j la longueur du mot. Il est facile de voir que ces mots sont les plus tôt et tard possibles étant donné i et j.

Quelques propriétés, avec $\forall_{x \in [0,1]} |early|_x = |late|_x = |a|_x$:

– $\forall i [early \sqcup early]_i = [early]_i$.
– $\forall i [early \sqcup a]_i = [a]_i$.
– $\forall i [late \sqcup late]_i = [late]_i$.
– $\forall i [late \sqcup a]_i = [late]_i$.

La taille maximale pour la longueur d'un buffer étant donné deux horloges ayant le même débit est bornée par la distance maximale qu'il y a entre deux occurences de 1, supposons que nous avons deux horloges avec les patterns : $early(i, j) = a$ et $late(i, j) = b$ qui sont les pires cas au niveau de cette distance, alors la borne maximale pour la longueur du buffer, supposons que nous avons *normalisé* ces horloges, c'est à dire $|a| = |b|$ et $|a|_1 = |b|_1$ alors nous avons : $Max_{buffer} = |a| - |a|_1 = |b| - |b|_1$.

Définition 10 (*Synchronisabilité* $\bowtie$)**.** Maintenant, nous introduisons un des aspects principal du N-Synchrone appelé la *Synchronisabilité* qui est la conjonction de la condition d'ordre partiel vue auparavant et l'équivalence de débit entre deux horloges : étant donné deux horloges a et b nous avons :

$\frac{|a|_1}{|a|} = \frac{|b|_1}{|b|}$ que nous noterons aussi $rate(a) = rate(b)$. À cause de la relation d'ordre nous savons qu'il existe une distance *bornée* entre le nombre d'occurences de 1 dans a et b.

Nous disons que deux mots infinis a et b sont *synchronisables* noté $a \bowtie b$, si et seulement s'il existe $dist, dist' \in \mathbb{N}$, tel que $a \preceq 0^{dist} b$ et $b \preceq 0^{dist'} a$ et si la condition de débit est vérifiée.

Buffer N-Synchrone Le stockage de ces occurences est effectué grâce à une FIFO (que nous appelerons Buffer N-Synchrone), l'action d'enfilage (*ENQUEUE*) est activée par l'horloge du producteur et le défilage (*DEQUEUE*) par celle du consommateur, il fonctionne comme suit :

- Si la FIFO est *vide* et que nous avons à la fois *DEQUEUE* et *ENQUEUE* alors nous faisons transiter la donnée directement.
- Autrement si *DEQUEUE* alors émission de la donnée.
  - Si *ENQUEUE* alors nous enfilons la donnée.
- Autrement si *ENQUEUE* alors nous enfilons la donnée.

Définition 11 (Normalisation d'horloge ($\leftrightsquigarrow$))**.** La *normalisation* d'horloge prend deux horloges *synchronisables* a, b et trouve le plus petit commun multiple $ppcm$ entre la longueur de leurs mots binaires associés. Nous concaténons ensuite chacun des mots avec lui-même jusqu'à obtenir la longueur voulue ($ppcm$). Nous notons cette normalisation de la longueur de deux mots de la manière suivante : $a \leftrightsquigarrow b$.

Compléments au N-Synchrone

Théorème 18. *La condition d'ordre pour la synchronisabilité dans la théorie N-Synchrone n'est pas significative. Il existe toujours pour n'importe quel mot a,b avec* $|a|_1 = |b|_1$ *et* $|a| = |b|$ *une paire de distances* $dist, dist' \in \mathbb{N}$ *(taille de buffer N-Synchrone) entre les occurences de* 1 *des mots a et b, telle que* $a \preceq c$ *et* $b \preceq c$ *où* $\exists c$ *tel que* $|c| = |a| = |b|$ *et* $|c|_1 = |a|_1 = |b|_1$.

Démonstration. Il suffit de construire c en appliquant $\forall i \in [1; |a|]\ [a \sqcup b]_i = [c]_i$, il faudra alors 2 buffers N-Synchrones à la sortie de a et b. □

Ainsi seulement le fait que les 2 horloges ont le même débit égal est suffisant pour disposer d'une synchronisabilité "relaxée".

Si nous avons à la fois de l'équivalence de débit et de l'ordre de précédence alors un seul buffer N-Synchrone est nécessaire sur la partie dominée grâce à la relation d'ordre.

Définition 12 (Rotation)**.** Une *rotation à gauche* $\circlearrowleft$ (et une *rotation à droite* $\circlearrowright$) d'un mot *NSynchrone w* s'applique sur la phase *periodique* de *v* de la manière suivante :

$v' = v \circlearrowleft offset$ tel que $\forall_{i\in[1;|v|_1]}\ [v']_i = [v]_i - offset\ \%|v|$

(et respectivement $v' = v \circlearrowright offset$ tel que $\forall_{i\in[1;|v|_1]}\ [v']_i = [v]_i + offset\ \%|v|$).

Où $offset \in [0;\infty]$ est l' *offset* de la rotation et % l'opérateur *modulo* usuel. La notation $[x]_i$ a été introduite dans la définition 6.

Par exemple : $010100 \circlearrowleft 1 = 101000$ et $010100 \circlearrowright 1 = 001010$.

Définition 13 (Équivalence d'ordonnancement ($\equiv^{\circlearrowright}$))**.** Équivalence rotationnelle d'ordonnancement : $schedule_a \equiv^{\circlearrowright} schedule_b$ si $\exists index \in \mathbb{N}$ tel que $schedule_a \circlearrowright index = schedule_b$ avec $schedule_a \bowtie schedule_b$ et $schedule_a \nleftrightarrow schedule_b$ où $schedule_{a,b} \in \mathbb{B}^*$

L'équivalence d'ordonnancement permet de vérifier si nous disposons du même ordonnancement modulo rotation.

Algorithme Pour vérifier simplement l'équivalence d'ordonnancement, nous appliquons la rotation *n* fois où *n* est la longueur du mot et nous effectuons un *XOR* sur les deux mots tel que si le résultat du *XOR* est 0...0 alors les deux mots sont équivalents modulo rotation.

Taille de FIFO Dans ce paragraphe nous détaillons plus formellement comment calculer la taille de la FIFO nécessaire entre deux entités N-Synchrones.

Pour ce faire nous définissons tout d'abord une notion de "quantité" d'un sous-mot N-Synchrone qui est simplement la somme des occurences de 1 du mot pour une longueur donnée. Cette quantité est définie formellement de la manière suivante : $quantity(a,i) = \Sigma_{j=0}^{j=i} quantity(a,j)$ avec $quantity(0,1) = 0$, $quantity(1,1) = 1$ où a est le mot N-Synchrone, $i \in [1;|a|]$ l'index considéré.

Par exemple, $a = (0110)$, $b = (1001)$, $quantity(a,2) = 1$, $quantity(b,2) = 1$, $quantity(a,3) = 2$, $quantity(b,3) = 1$.

Ensuite, nous définissons une notion de "distance" entre une paire de sous-mots N-Synchrones qui est la valeur absolue de la différence entre les quantités des sous-mots considérés (nous supposons que les mots sont égaux en longueur). Cette distance est définie comme suit : $distance(a,b,i) = quantity(a,i) - quantity(b,i)$ où a,b sont des mots N-Synchrones et $i \in [1;|a|]$ ($|a| = |b|$) l'index considéré.

Par exemple, $a = (0110)$ et $b = (1001)$, $distance(a,b,1) = -1$, $distance(a,b,2) = 0$, $distance(a,b,3) = 1$, $distance(a,b,4) = 0$.

Enfin, nous pouvons définir la taille maximale de la FIFO qui est simplement la distance maximum quel que soit l'index considéré d'une paire de sous-mots, qui est définie ainsi : $taille_{maxFIFO} = \forall_{i=1}^{i=|a|=|b|} MAX(positive(dist(a,b,i)))$ où la fonction $positive(i) = i$ si $i \in [0;\infty]$ et $i = 0$ sinon.

Par exemple, $a = (0110)$ et $b = (1001)$, nous avons dans ce cas $taille_{maxFIFO}(a \rightarrow b) = taille_{maxFIFO}(b \rightarrow a) = 1$. Nous avons besoin d'une FIFO à la sortie de a (et b respectivement) de taille 1.

Si nous nous plaçons dans le cas de la *synchronisabilité* du N-Synchrone, il y a un mot qui "domine" l'autre : il suffit d'une FIFO du côté dominé. Si a domine b, alors $dist(a,b,i) \geq 0$. Dans le cas général nous aurons besoin de deux FIFOs, pour calculer leurs tailles respectives, nous évaluons $dist(a,b,i)$ et ensuite $dist(b,a,i)$.

Le modèle N-Synchrone élargit le modèle synchrone en généralisant le principe d'horloge à une représentation avec des mots binaires infinis. Ces mots binaires infinis sont composés par un premier mot binaire décrivant une phase d'initialisation optionnelle et un mot binaire répété périodiquement à l'infini. Chaque composant N-Synchrone a un débit qui est le rapport entre le nombre d'occurences de 1 et la longueur du mot périodique répété à l'infini. Le problème est que le nombre de ressources nécessaires et suffisantes pour synchroniser des composants N-Synchrones peut être non-borné dans le cas général : une condition de *synchronisabilité* nécessaire afin que le nombre de buffers soit borné est que tous les modules N-Synchrones connectés entre eux disposent du même débit. La condition de synchronisabilité du N-Synchrone dit en l'espèce qu'il existe une abstraction d'un instant global au sens synchrone composé de différentes répétitions d'un mot binaire périodique infini.

4.1.3 Ordonnancement des Marked Event Graphs

Des études sur les réseaux LID et les Marked Event Graphs (MG) essaient de fournir des règles d'exécution synchrones aux réseaux, dans l'idée que tous les nœuds de calcul s'exécutent dès que faisable, et simultanément si possible. Leurs cadres et motivations divergent sur différents points :

– Dans la théorie LID [63, 62] les places de stockage sont remplacées par les *stations de relais*, et les nœuds de calculs sont encapsulés par des *shells*. Le but de ces composants, qui sont des éléments synchrones additionnels [14], permettent d'implanter l'ordonnancement dynamique en ligne : ils régulent le trafic des données au point que jamais plus de deux jetons s'accumulent dans n'importe quelle station de relais. Les nœuds de calcul ont besoin d'avoir sur toutes leurs entrées au moins un jeton, mais aussi suf-

fisamment d'espace libre dans les stations de relais placées sur les sorties. Dans l'exemple de la figure 4.1 (c), par exemple, la seconde étape d'initialisation la station de relais grisée nécessite de conserver deux jetons (pendant que le jeton sur l'arc sur la droite est en cours de transit).

– Des recherches conduites sur les MGs essaient d'obtenir un ordonnancement statique répétitif , basé sur le fait que l'exécution synchrone donne un comportement déterministe, ce dernier se répète au bout d'une période donnée à cause de la finitude des répartitions de jetons. Mais l'ordonnancement ne fait que peu attention à la distribution des jetons entre les places et donc à la taille des mémoires tampons (FIFOs). Les fondations de la théorie des ordonnancements statiques et *k*-périodiques pour les MGs se trouvent dans [19, 8]. Dans [19] les auteurs appellent cela le *Problème Central Répétitif* (CRP).

En fait, nous pouvons voir aisément que l'ordonnancement dynamique gouvernant les règles d'évaluation synchrones des LIDs et MGs sont des comportements *ultimement répétitifs*, qui sont amenables à un ordonnancement statique qui peut être calculé hors-ligne. Entre autres, l'absence de contrôle dans ces systèmes permet le déterminisme. Chaque configuration instantanée amène à une configuration suivante unique en exécutant tous les nœuds de calculs et de transports possibles (ici nous appelons une configuration une allocation de jetons sur les mémoires tampons ou les stations de relais) ; l'ensemble des configurations possibles est fini (souvenons nous que nous avons supposé le cadre des composantes fortement connexes, qui garantissent un invariant du nombre de jetons présents sur chaque boucle) ; alors le système arrivera sur un état déjà visité (une borne) après une phase d'initialisation finie. Ainsi, le comportement d'un système synchrone LID ou MG consiste en une phase transitoire d'initialisation, suivie d'une phase stationnaire périodique. La même période s'applique à tous les nœuds de calculs et de transport du système, et à l'intérieur de la période le nombre d'exécutions est le même pour tous les nœuds (cela se nomme la *périodicité*) ; ainsi nous pouvons parler de la période et la périodicité de graphes.

Plus formellement, soit un mot N-Synchrone $a = u(v)$, $|v|$ est la période et $|v|_1$ la périodicité. Le rapport $|v|_1/|v|$ est appelé débit. Un circuit a un débit qui est le rapport entre le nombre de jetons et le nombre de transitions. Dans un graphe, un circuit est dit critique s'il est de débit minimal, il est non-critique autrement.

Notons qu'en général il n'y a pas de garantie que le débit de calcul corresponde à celui indiqué dans les données sur les latences locales fournies par l'utilisateur. Les données traversant les circuits non-critiques peuvent avoir à attendre pour d'autres passant sur des routes plus lentes avant de rejoindre les nœuds de calculs. En un sens, les latences indiquées fournissent un seuil sur les latences effectives, au moins sur les circuits non-critiques.

[8] fournit des bornes sur les tailles de la période et la périodicité. Une borne pour la périodicité k a été établie comme la périodicité de G^c, où G^c représente la restriction de G à ses circuits critiques. La périodicité de G^c est égale au *ppcm* (plus petit commun multiple) de la périodicité de chaque composante fortement connexe de G^c. La périodicité de chaque composante fortement connexe de G^c est égale au *pgcd* (plus grand commun diviseur) du nombre de jetons contenus dans chaque cycle. La période peut être calculée de la même manière, considérant le nombre de latences présentes sur chaque cycle au lieu du nombre de jetons. Dans la figure 4.1(c), le graphe est 3 périodique de période 5. [8] fournit des bornes sur les tailles des périodes et périodicités.

Malheureusement, contrastant avec ces résultats probants peu de choses sont connues sur la longueur de la phase d'initialisation amenant à cette phase stationnaire.

4.1.4 Représentation explicite des Ordonnancements

Nous allons utiliser une notation syntaxique explicite afin de manipuler les ordonnancements comme objets du modèle, nous reprenons l'idée de notation issue de la théorie des processus N-synchrones [33], où elle est utilisée afin de typer des programmes avec leurs ordonnancements (c'est à dire, une expression représentant la séquence des instants où le système est exécuté). Nous l'employons d'une manière bien différente, en fournissant l'ordonnancement à chaque nœud de calcul ou de transport dans le réseau. Comme dans leurs travaux, nous nous concentrons sur les ordonnancements et comportements périodiques.

Définition 14 (Ordonnancements)**.** Un *ordonnancement* pour un graphe est une fonction $N \rightarrow w_N$ affectant un mot infini $w_N \in \{0,1\}^\omega$ à tout nœud de calcul ou de transport du réseau. L'intuition est que l'ordonnancement oblige l'exécution lors des instants où un 1 est présent et d'inactivité lorsque il y a un 0. Un ordonnancement est dit périodique lorsque le mot w est de la forme $w = u(v)^\omega$ où $u, v \in \{0,1\}^\star$. u est appelé la partie initiale, et v la partie périodique. Nous appelons la longueur de v (notée $|v|$) la période de w, et le nombre de 1 dans v (notée $|v|_1$), la périodicité de w.

Supposons que pour tous les nœuds de calcul et transport, tous les w_N ont des périodes et périodicités identiques, nous notons respectivement p et k la période et la périodicité du réseau. Le *débit* r peut être défini comme $\frac{k}{p}$. Un ordonnancement est admissible si les w_N ont une relation mutuelle qui respecte la sémantique de la règle d'exécution du modèle (depuis n'importe quelle configuration initiale des jetons il y a un et un seul ordonnancement admissible, suivant la sémantique déterministique synchrone).

La figure 4.1(d) illustre l'utilisation de la notation pour les ordonnancements. Toutes les séquences d'exécution sont décrites, telles que nous les aurions obtenues en faisant "tourner" cet exemple. Comme attendu les parties périodiques de l'ordonnancement sont de longueur 5 (la période) et contiennent chacune 3 occurences du symbole 1 (la périodicité). L'ordonnancement périodique permet en plus de ces figures, de visualiser la distribution des jetons à l'intérieur de la période de chaque nœud. Idéalement, si le système est bien-équilibré du point de vue des latences, alors l'ordonnancement d'un nœud de calcul donné doit être exactement le même que son prédécesseur décalé d'une position (modulo rotation). Mais lorsque les données n'arrivent pas au même instant pour un nœud donné, certaines valeurs doivent être bloquées à l'entrée. Dans notre exemple cela arrive seulement sur le nœud de calcul positionné au plus haut de la figure. Nous préfixons certains 0 d'inactivité par des marques indiquant si l'inactivité est causée par un jeton à l'entrée de droite (′) ou de gauche (‘).

4.2 Ordonnancement Statique de systèmes LID et Égalisation des Latences

4.2.1 Égalisation d'un graphe

Maintenant nous allons revenir sur le cas général : nous introduisons un ordonnancement statique périodique très spécifique, obtenu par insertion de latences additionelles sur les sections de transport qui sont plus "rapides" que les autres. Le but est d'essayer et de faire que les différents chemins de transports soient aussi uniformes que possible en terme de latence, de telle manière que les données arrivent simultanément à leur destination commune. Cette transformation ne doit pas pénaliser le débit global du réseau qui fonctionne originellement au débit des cycles les plus lents.

Ces latences supplémentaires sont virtuelles, dans le sens où elles ne correspondent pas à des demandes, des contraintes liées à la physique. Dans un cadre méthodologique elles peuvent être utilisées afin de "redesigner" certains composants en agissant sur leur longueur de pipeline et/ou en utilisant des portes de plus petite surface par exemple.

Malheureusement, comme notre exemple de la figure 4.1(b) montre, l'égalisation exacte des latences n'est pas toujours possible avec des latences entières. La boucle à gauche a un débit de $2/3$, et la plus lente, celle de droite a un débit de $3/5$. Mais ajouter une latence virtuelle sur la boucle de gauche diminuerait trop le débit à $\frac{2}{3+1} = 1/2$, qui est strictement inférieur au plus lent. Notre approche consistera à insérer d'abord autant de latences entières que possible, et ensuite d'ajouter des

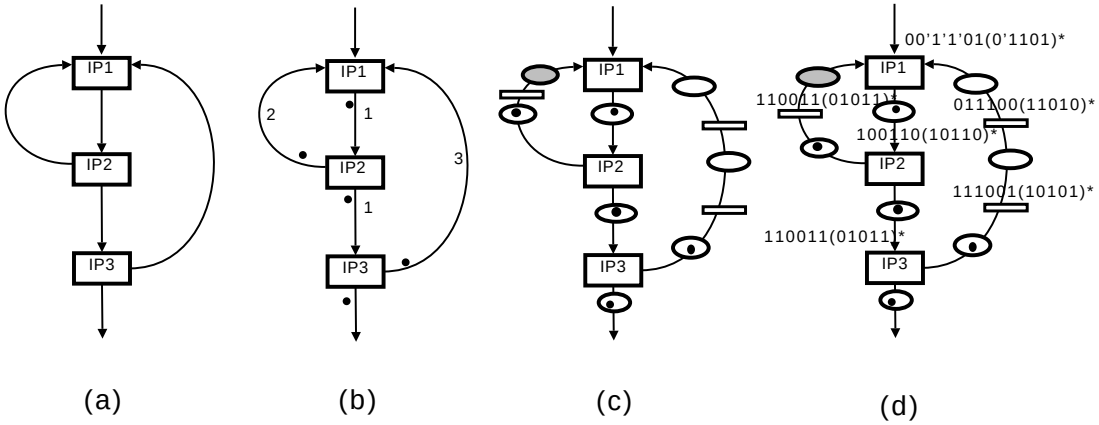


FIG. 4.1 – (a)CN, (b)marquage et latences, (c)RSs, (d)ordonnancement explicite avec égalisation

registres fractionnaires (FR) qui sont des éléments synchrones permettant lorsqu'ils sont positionnés de manière appropriée de corriger les écarts pour atteindre l'égalisation. Les registres fractionnaires conservent temporairement un jeton de temps en temps. Nous avons longtemps cru que la complétion préléminaire avec des entiers, puis l'effet de "lissage" lors de la phase d'initialisation étaient les instruments pour distribuer uniformément les jetons. Si cela était le cas, les FRs seraient réduits à de simple registres court-circuitables (voire la section suivante pour les problèmes de correction).

Dans notre exemple de la figure 4.1 (d) un registre fractionnaire est nécessaire à la place grisée dans la boucle la plus à gauche, pour conserver un jeton chaque fois que le nœud de calcul placé le plus haut indique un $0'$ sur l'ordonnancement, signifiant qu'à cet instant le canal de communication sur la droite ne peut pas encore délivrer sa valeur.

Ensuite, nous allons décrire les différentes étapes algorithmiques utilisées dans le processus d'égalisation. Ces étapes sont : *déterminer le débit global atteignable* ; *calculer le nombre maximal de latences entières que nous pouvons ajouter* (ajouter les éléments correspondant à la description) ; *calculer les ordonnancements transitoires et stationnaires* (par une construction linéaire de l'espace d'états) ; *ajout des registres fractionnaires*. Ces étapes s'appuient la plupart du temps sur des techniques bien connues sur les graphes ou la programmation linéaire. Dans le futur d'autres étapes algorithmiques (optionnelles) peuvent être étudiées, afin en particulier de raccourcir la séquence d'initialisation, en la rendant asynchrone.

Nous décrivons maintenant les étapes successives :

Évaluation globale du débit. Nous avons besoin de calculer le débit global atteignable, qui est le débit le plus lent (noté *R*) entre les différentes boucles. Pour cela, nous n'avons pas besoin techniquement d'énumérer tous les cycles élémentaires, et d'utiliser un algorithme résolvant le problème bien connu du *Minimal Cost-to-Time Ratio Cycle problem* [51, 36]. Néanmoins, nous avons besoin de tous ces cycles dans l'étape suivante.

Insertion des latences entières. Ce problème est résolu par l'utilisation de techniques de programmation linéaire. Le système d'équations linéaires est construit de manière à exprimer tous les cycles élémentaires, avec des variables supplémentaires sur les arcs pour ajouter des latences, qui doivent être maintenant du débit *R*, le débit global trouvé auparavant. Les équations sont formées lors de la phase précédente d'énumération des cycles. Une contrainte additionnelle est entrée pour que le solveur minimise le nombre de latences ajoutées de façon à factoriser au mieux ces dernières.

Plutôt que d'étudier une solution rationnelle et après d'en extraire un entier approximatif pour la valeur des latences, la forme particulière des systèmes d'équations permet l'utilisation d'un algorithme direct *glouton*, permettant d'ajouter incrémentalement des latences entières jusqu'à la complétion. Cela fut confirmé par notre prototype d'implantation.

L'exemple suivant de la figure 4.2 montre que notre complétion d'entiers ne garantie pas que tous les cycles élémentaires aient un débit suffisamment proche du minimal.

Ainsi dans cet exemple aucune insertion de latence entière peut amener tous les cycles à un débit assez proche du débit global. Le débit global est de $\frac{3}{16}$, donné par le cycle interne, aucune latence entière ne peut être ajoutée au débit de $\frac{1}{5}$ (car $\frac{1*16}{6*16} < \frac{6*3}{6*16}$). À la place quatre latences fractionnelles doivent être ajoutées (sur chaque arc de poids 1). Mais ici c'est à cause du fait qu'un cycle "touche" le plus lent en différents points distincts. Une explication plus formelle est que si nous construisons la matrice où chaque ligne est l'équation du débit de chaque cycle (nombre de jetons sur nombre de latences associé à chaque arc du cycle), il s'avère qu'il existe aucune variable libre dans ce système, c'est à dire que le déterminant de la matrice est nul, il n'y a pas de solution, il n'est pas possible de rajouter de latence entière. Néanmoins, il existe une famille de graphes où l'égalisation avec des latences entières est toujours possible, par exemple celle où tout cycle dispose du même nombre de jetons et où la précédente matrice dispose d'un déterminant non nul : il s'agit en général du cas où nous avons dérivé une spécification synchrone en une spécification latency-insensitive.

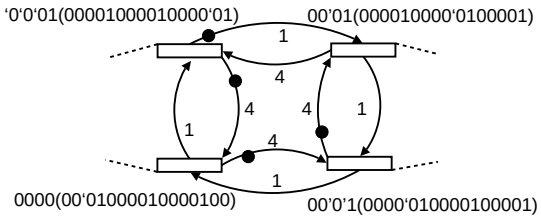


FIG. 4.2 – Égalisation entière impossible en général

Calcul de l'ordonnancement (utilisant une construction de l'espace d'états) Dans le but de calculer les ordonnancements explicites des phases transitoires et stationnaires nous avons besoin de *simuler* le comportement du système. Nous avons besoin aussi de conserver l'état visité, comme un critère de terminaison pour la simulation lorsque qu'un état déjà visité a été atteint. Le but est de construire (simultanément ou dans une seconde phase) les ordonnancements des nœuds de calculs, en incluant les marques $'$ et ‘, de façon à déterminer où le résidu fractionnaire de latence doit être inséré. Dans une exécution synchrone chaque état aura seulement un successeur, et ce processus s'arrêtera dès qu'un état déjà rencontré aura été atteint. Le problème principal ici consiste dans la complexité de l'espace de représentation. Dans une approche naïve chaque place peut contenir T jetons, où T est la somme minimale des jetons sur tous les cycles élémentaires utilisant cette place. Mais avec les latences supplémentaires, nous pouvons utiliser une modélisation LID avec des stations de relais et le mécanisme de contre-pression. Après chaque place peut contenir au plus deux jetons, encodés par deux valeurs booléennes. Alors l'état global est encodé avec $2n$ variables booléennes, où n est la somme de toutes les latences portées par les arcs. Plusieurs types de simplifications de l'espace d'états à la manière du model-checking symbolique en utilisant des BDDs sont possibles à cause du fait qu'entre deux valeurs décrivant un état d'une station de relais, une station de relais est remplie seulement si une autre l'était aussi.

Latences Fractionnelles Dans un système idéalement égalisé, les ordonnancements distincts de nœuds de calcul/transport sont en relation : l'ordonnancement du “prochain” nœud et celui du “précédent” décalé d'un coup sur la droite (et les premières valeurs de l'ordonnancement dans la phase initiale ne le sont pas). Après nous calculons les ordonnancements effectifs, nous pouvons vérifier si c'est le cas. Si cela ne l'est pas, les registres fractionnels supplémentaires seront insérés juste après le registre usuel déjà inséré entre les nœuds. Ce registre fractionnaire retardera quelques jetons (mais pas tous). Nous introduirons un modèle formel de

nos registres fractionnels dans la prochaine sous-section. Le diagramme bloc de son interface est fournis dans la figure 4.3.

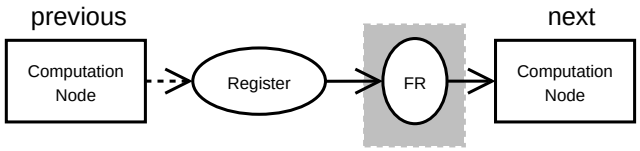


FIG. 4.3 – Insertion d'un registre fractionnaire dans le réseau.

Implémentation Nous avons implanté avec Jean-Vivien Millo, cet algorithme dans un outil appelé KPASSA (K-Periodic As-Soon as possible Scheduling and Analysis) permettant de décrire de manière graphique une structure de graphe, de lui associer un placement initial des jetons, d'appliquer les différentes phases de l'algorithme et de générer sous forme textuelle ou graphique des résultats tel que le nombre de latences virtuelles ajoutées, l'ordonnancement statique trouvé avec sa phase d'initialisation, le placement et la taille des registres fractionnaires après simulation du système. L'ensemble du logiciel est écrit en Java utilisant la bibliothèque MascoptLib écrite par le projet Inria Mascotte, il y aussi un lien vers deux solveurs connus CPLEX et GLPK. Nous présentons quelques résultats pour étudier la complexité en temps de notre algorithme (simulé sur un dual xeon 3 Ghz, 8Go RAM, Linux, JDK 1.5.11 et GLPK).

Noeuds	Cycles	LP	Simulation	Total
12	23 (78ms)	5ms	1137ms	1311ms
20	12 (65ms)	3ms	6095ms	6336ms
40	2295 (4480ms)	545ms	35434ms	43481ms
81	2296(8088ms)	664ms	15813ms	30151ms

Cette implémentation a confirmé le fait que le système d'équation fourni au solveur est relativement facile à résoudre, la plupart du temps étant passé dans la phase de recherche des cycles dans le graphe qui est de complexité quadratique et surtout la simulation. Le temps de simulation dépend à la fois du nombre de nœuds et d'arcs mais surtout du placement initial des jetons (la simulation la plus courte en temps est celle dans laquelle le placement initial est le même que dans le régime k-périodique). Ceci explique la baisse en temps de la simulation entre les deux dernières lignes du tableau.

Initialisation Optimisée Jusqu'à présent nous avons seulement considéré le cas où tous les composants s'exécutent dès que possible (ASAP). Quelques fois retarder certains calculs ou transports dans la phase initiale permettent d'atteindre plus rapidement la phase stationnaire ou une autre phase stationnaire distincte qui peut être mieux distribuée que l'ordonnancement originel. Considérons l'exemple de la figure 4.1 (c) la possibilité d'exécuter tout seul le nœud de transport en bas à droite (celui qui a un arc de retour) au premier instant. En effectuant cela nous atteignons immédiatement le régime stationnaire (dans sa dernière étape d'itération).

Les phases d'initialisation peuvent nécessiter un nombre important de ressources de stockage temporaires, qui ne seront plus du tout utilisées lors de la phase stationnaire. Fournir une séquence d'initialisation courte et utilisant peu de ressources de stockage est un challenge. De plus, minimiser le nombre de valeurs initiales présentes sur les différents cycles est un problème NP complet connu sous le nom de Minimum Feedback Arc Set, un algorithme efficace pour le résoudre est décrit dans ce papier [39].

4.2.2 Registre Fractionnaire

Nous allons maintenant décrire formellement le registre fractionnaire, à la fois comme un circuit synchrone dans la figure 4.4(b), son automate à états finis correspondant dans la figure 4.4(a) et le chemin de données associé dans la figure 4.4(c).

L'interface du registre fractionnaire consiste en deux fils d'entrée *val_in* et *hold*, et d'un fil de sortie *val_out*. Son état interne consiste en un registre *catch_reg*. Le registre peut être utilisé pour "kidnapper" le jeton (et sa valeur dans un cas réel) pour un cycle d'horloge lorsque *hold* est présent. Nous notons $pre(catch\_reg)$ la valeur (boolénne) du registre calculé lors du précédent cycle qui indique si le registre est occupé ou non.

Il est possible que le même jeton soit conservé plusieurs instants. Cependant il ne doit pas y avoir de nouveau jeton arrivant, comme le registre fractionnaire ne peut stocker seulement qu'une valeur ; sinon cela causerait un conflit. Il est aussi possible qu'une séquence complète de jetons soit décalée d'un instant. C'est à dire que chaque jeton/valeur doit laisser l'élément présent être consommé le coup suivant ; autrement il y aurait un problème de correction.

Exprimé formellement, lorsque $hold \wedge pre(catch\_reg)$ est vrai alors *val_in* est aussi vrai, dans ce cas la nouvelle valeur associée au jeton entre et la valeur précédente sort (par consistance de l'ordonnancement les nœuds de calculs qui consomment doivent donc être actifs), ou *val_in* n'est pas vrai, dans ce cas le jeton courant reste (et réciproquement le nœud de calcul doit être inactif). De plus les deux conditions suivantes sont nécessaires :

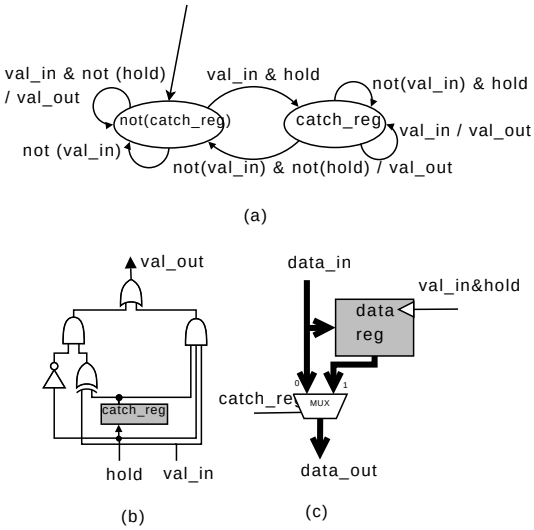


FIG. 4.4 – L'automate, interface et chemin de données du registre fractionnaire

[$hold \Rightarrow (val\_in \vee pre(catch\_reg))$:] si rien ne doit être conservé alors l'ordonnancement n'essaie pas de faire cela ;

[$(val\_in \wedge pre(catch\_reg)) \Rightarrow hold$:] autrement les deux jetons peuvent traverser l'élément et passer à la sortie simultanément.

Le comportement du registre fractionnaire correspond aux deux exemples suivants :

[$catch\_reg = hold$:] le registre est utilisé seulement lorsque l'ordonnancement en fait la demande ;

[$token\_out = token\_out_1 \vee token\_out_2$:]
- $token\_out_1 = val\_in \oplus pre(catch\_reg) \wedge \neg hold$.
- $token\_out_2 = val\_in \wedge pre(catch\_reg) \wedge hold$.

soit une nouvelle valeur passe directement au travers, ou une ancienne est chassée par une nouvelle qui prend sa place.

Notre problème principal de design est maintenant de générer le signal *hold* exactement lorsqu'il est nécessaire en respectant les contraintes précédentes. Or, il se trouve qu'il peut être généré en fonction des ordonnancements des nœuds de calculs (transports) source et de calcul, de façon à lier le premier avec le dernier.

Considérons encore la figure 4.3, nous appelerons w l'ordonnancement du *précédent* nœud source, et w' l'ordonnancement du nœud *suivant* cible. Après que le registre normal est retardé, les jetons sont produits à l'entrée du registre fractionnaire avec l'ordonnancement $0.w$ (décalage sur la droite d'un instant). Le registre

fractionnaire doit conserver le jeton exactement lorsque la k^{ieme} occurence de 1 est présente à son entrée et non pas la k^{ieme} occurence du nœud cible qui doit consommer le jeton. En d'autres termes le registre fractionnaire resynchronise les entrées et les sorties, qui ne peuvent être au delà d'une occurence d'activité. Cette dernière propriété est vraie si les ordonnancements sont calculés en utilisant l'approche LID avec les stations de relais, qui n'autorisent pas plus d'un jeton de plus en addition du registre normal entre les nœuds de calcul et transport.

Plus formellement, la propriété devient : $hold(n) = 1$ *ssi* $|0.w_n|_1 \neq (|w'_n|_1 - |w'_0|_1)$: à un instant donné n nous devons "kidnapper" une valeur si le nombre d'occurences de 1 jusqu'à cet instant n sur le nœud précédent est différent du nombre d'occurences sur le nœud suivant. Plus précisément, le terme $-|w'_0|_1$ prend en compte l'activité d'initialisation du nœud cible, non causée par la propagation des jetons depuis le nœud source, qui auraient du être enlevés.

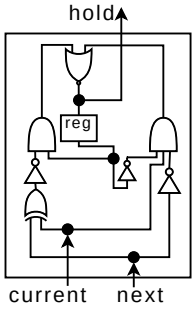


FIG. 4.5 – Implémentation de *hold*.

La figure 4.5 montre une implantation possible pour calculer *hold* depuis les signaux des ordonnancements explicites des nœuds source (entrée *current*) et destination (entrée *next*) fournis en entrées. Bien sûr il est possible d'utiliser la technique des registres à décalage comme dans le cas des nœuds de calcul ou transport.

4.2.3 Composition de systèmes ordonnancés

Le but de cette section est d'étudier comment des blocs plus "larges", obtenus en tant que composants synchrones interconnectés par le protocole LID optimisé par les informations issues de la phase d'ordonnancement statique, peuvent être interconnectés dans un environnement avec son propre débit.

Nous appellerons un réseau élémentaire ordonnancé statiquement une SSIP (Statically Scheduled IP) qui est notre bloc élémentaire.

Supposons qu'un nombre de tels systèmes LID ont été maintenant et indépendemment ordonnancés statiquement, nous voulons étudier comment les composer

dans un système plus large de systèmes, et ce sans altérer l'ordonnancement statique précédent si possible. Bien sûr le système global fonctionnera au débit le plus lent des composants.

Nous affectons à chaque port d'entrée et sortie un ordonnancement, en utilisant la notation N-Synchrone précédente. Il n'est pas nécessaire de connaître les ordonnancements internes.

Comme nous l'avons introduit précédemment les SSIPs sont construites indépendamment les unes des autres, avec leur débit propre ($\frac{k}{p}$). La composition de ces SSIPs avec d'autres, composants GALS ou synchrones fonctionne au débit du plus lent. Toutes les autres SSIPs doivent pouvoir ralentir sans pour autant modifier leur comportement interne. Cela peut être facilement effectué "théoriquement" en supposant qu'il est possible d'effectuer un "clock-gating" géré par l'état des FIFOs présentes sur les entrées et sorties. Si une FIFO en entrée est vide ou que la FIFO en sortie est pleine, cela signifie que la SSIP fonctionne "trop vite" par rapport à son environnement.

Le débit "réel" d'une SSIP est le produit de son rapport périodicité sur période multiplié par la fréquence d'horloge propre au système. Ainsi lorsque nous allons composer des SSIPs entre eux les performances dépendront du/des SSIPs ayant les débits réels les plus lents comme dans le cadre synchrone.

Nous supposons qu'il est possible de construire des FIFOs multi-horloges telles que celles décrites dans ce papier [32] permettant d'assurer l'interconnection entre les SSIPs avec des vitesses relatives différentes.

Propriété 3. *Toute interconnection entre des SSIPs dispose de FIFOs de taille bornée.*

Démonstration. Comme nous l'avons explicité auparavant, chaque SSIP a la même périodicité et période quel que soit le nœud considéré donc les entrées/sorties portent les mêmes valuations dans l'abstraction vers SDF, amenant à un graphe SDF uniforme où il existe toujours un ordonnancement avec des FIFOs de taille bornée.
L'abstraction est construite de la manière suivante : Nous calculons les débit "réels" des entités, trouvons le débit le plus lent qui sera alors abstrait dans un graphe SDF uniforme et ce nœud aura sur ses entrées/sorties la valeur 1. Nous calculons pour tous les autres le plus petit commun multiple entre son débit "réel" et celui du plus lent et associons cette valeur sur les entrées/sorties du nœud associé. □

Effectuer une simulation sur l'abstraction SDF uniforme permet de calculer une borne maximale sur la taille des FIFOs nécessaires entre les SSIPs lors du régime périodique. Cette borne est inutile, car en général les SSIPs ne produisent pas

toutes leurs valeurs en “raffale” et de même lors de la consommation des données. Cela implique qu’il existe des formes particulières des ordonnancements permettant de minimiser la taille des FIFOs d’interconnection entre les SSIPs, l’idéal étant que la FIFO soit au plus de taille unitaire ou nulle.

Pour calculer la taille minimale des FIFOs, il est alors nécessaire de simuler la composition des SSIPs en utilisant leurs ordonnancements explicites ainsi que la phase d’initialisation. La transformation est alors la suivante : nous dérivons à partir des horloges fournies, une horloge plus rapide qui est le plus petit commun multiple de ces dernières. Ensuite nous altérons les ordonnancements des SSIPs en fonction de cette horloge plus rapide de façon à obtenir le même débit “réel” que nous avions initialement dans chaque SSIP. L’idée est d’injecter un certain nombre de 0 avant (ou après) chaque symbole dans l’ordonnancement originel de façon à disposer du même débit “réel”. Ce nombre d’occurences de 0 rajouté à chaque symbole de l’ordonnancement originel est : $(horloge_{rapide}/horloge_{SSIP}) - 1$.
Schéma de preuve : nous posons que les débits sont égaux $k/p * horloge_{SSIP} = horloge_{rapide} * k/(p + (p * (horloge_{rapide}/horloge_{SSIP}) - 1))$, où k est le nombre d’occurences de 1 dans l’ordonnancement du SSIP et p la longueur de l’ordonnancement. Nous factorisons par k des deux côtés, nous effectuons ensuite un produit en croix, développons et simplifions.

Il reste à simuler le système transformé avec la même horloge rapide et les ordonnancements modifiés pour obtenir la taille des FIFOs nécessaires dans le système originel multi-horloge.

Sur la figure 4.6 (a) et (b) nous décrivons l’ordonnancement de deux SSIPs : l’une de débit $\frac{2}{3}$ avec une horloge à 300 Mhz, l’autre de débit $\frac{1}{2}$ avec une horloge à 400 Mhz. Le plus petit commun multiple entre ces deux horloges est une horloge à $4 * 300 = 3 * 400 = 1200$ Mhz. Nous construisons ensuite les ordonnancements afin d’obtenir le même débit “réel” : $\frac{2}{3*4}$ et $\frac{1}{2*3}$. Prenons le cas de (a), nous allons ajouter 3 occurences de 0 après chaque symbole de l’ordonnancement 101 initial (en utilisant la formule énoncée auparavant) nous obtenons alors l’ordonnancement 1.0000.0001.000 qui est représenté dans la figure 4.6 (c). Nous pouvons effectuer la même transformation sur l’ordonnancement de la figure 4.6 (b) et obtenons celui de la figure 4.6. Remarquons que la transformation des ordonnancements conservent les transitions des fronts montants, indiqué sur la figure 4.6 par des flèches en pointillés.

4.3 Résumé

Dans ce chapitre nous avons décrit de manière détaillée comment à partir d'un système LID nous pouvons obtenir un même système ordonnancé de manière statique tout en conservant les performances optimales en terme de débit du système. Cette technique d'ordonnancement statique est particulière, elle cherche à *égaliser* le plus possible les débits entre les différents circuits qui peuvent exister dans le système : pour se faire nous altérons alors la topologie en rajoutant des éléments de stockage supplémentaires. L' *égalisation* est une technique d'ordonnancement permettant de supprimer le protocole assurant la synchronisation dynamique. Intuitivement, le but est qu'en "égalisant" les latences des canaux de communication nous puissons synchroniser l'arrivée des données à tout nœud de calcul. Ce qui rend le protocole utilisé dans Latency Insensitive redondant et inutile. Historiquement, Casu et Macchiarulo dans [28] ont utilisé une technique d'ordonnancement appartenant à la famille des algorithmes de Software Pipelining permettant de supprimer ce protocole sans altérer la topologie tout en conservant les performances originelles du circuit en terme de débit. Carloni parallèlement a effectué des travaux sur la même contrainte de performance mais a utilisé une technique différente, il modifie la topologie en ajoutant des Relay-Stations sur les circuits non-critiques, de façon a les amener à une valeur proche des circuits les plus lents permettant ainsi de limiter fortement le recours au protocole de synchronisation nécessaire. Dans les 2 cas nous avons à faire face à un problème qui est dans $\mathbb{Q}$ et qui est partiellement résolu dans $\mathbb{N}$, ainsi dans le cas de Casu et Macchiarulo cela implique l'utilisation d'une horloge rationnelle qui a autant de phases que le mot périodique de la solution, dans le cas de Carloni il s'agit de l'utilisation du protocole de back-pressure. Nous avons proposé conjointement avec Robert de Simone et Jean-Vivien Millo, de conserver la technique d'ordonnancement statique cyclique tout en respectant les contraintes de performances initiales en terme de débit, tout en utilisant à la fois l'ajout d'éléments de stockage de données usuels mais en ajoutant aussi un élément particulier que nous appelons Registre Fractionnel qui permet de combler la partie fractionnaire nécessaire à la synchronisation. Nous combinons en fait à la fois les deux techniques introduites auparavant :

- alteration de la topologie sans affecter les performances du système.
- obtention d'un ordonnancement statique amenant aux performances optimales.
- pas besoin d'utiliser une horloge fractionnaire pour arriver à ces performances.
- utilisation d'un élément de stockage particulier nommé registre fractionnaire.
- simplification des stations de relais, suppression du protocole de back-pressure.

Ce chapitre a donné lieu à une publication dans la conférence internationale MEMOCODE en 2006 [16], une autre dans la conférence locale SAME en 2006 [15], et un article dans le Journal Eurasip en 2007 [13].

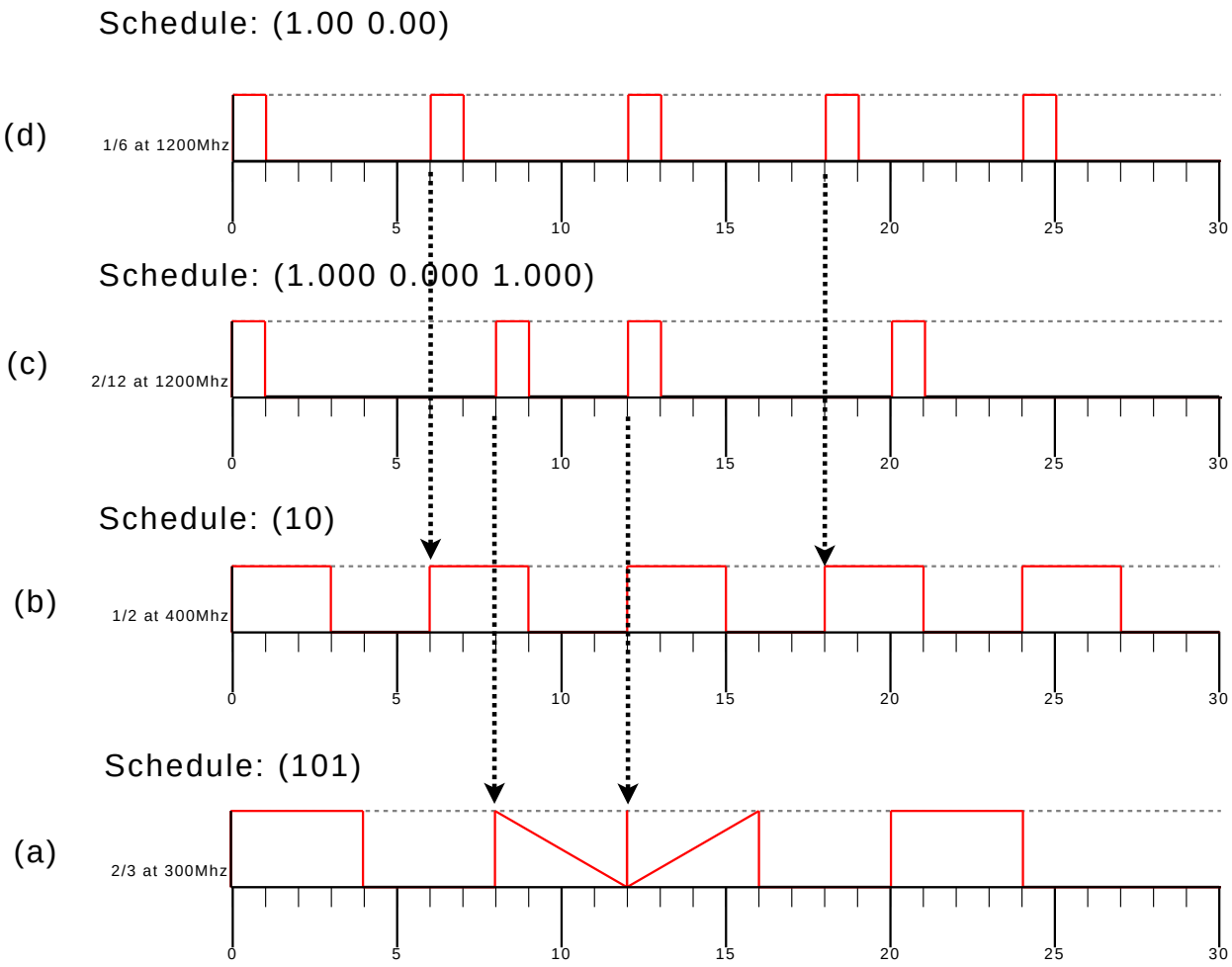


FIG. 4.6 – SDF Uniforme : normalisation d'horloge et modification ordonnancement

Chapitre 5

Introduction d'un contrôle sans conflit

Pour s'améliorer, il faut changer. Donc, pour être parfait, il faut avoir changé souvent. *Winston Churchill*

Dans ce chapitre, nous introduisons une extension des Marked Event Graphs avec une notion de contrôle sans conflit et déterministe s'inspirant des réseaux de Kahn. Dans ce modèle nous pouvons décider s'il existe un ordonnancement statique dont la taille des FIFOs est bornée, grâce à une abstraction effectuée sur le modèle Synchronous Data Flow.

5.1 Motivation

Jusqu'ici, nous avons discuté de modèles aux calculs répétitifs et réguliers sans alternative, sans choix, sans contrôle. L'absence de choix assure la confluence, c'est à dire que le fait d'exécuter une transition n'empêche pas d'en exécuter une autre plus tard. Néanmoins, il est possible d'introduire une dose limitée de choix "prévisible" en conservant cette propriété, il s'agit du sujet de ce chapitre. Nous avons vu que l'introduction de la latence ne fait que modifier l'ordre relatif des comportements par rapport à une spécification synchrone ou asynchrone.

Exemple motivant notre démarche Le problème inhérent des modèles synchrones, Marked Event Graph ASAP et Synchronous Data Flow est que le composant avec le débit le plus lent va limiter les performances globales du système.

La figure 5.1(a) illustre un modèle d'un traitement en "pipeline" où nous avons les nœuds de calcul de début et de fin du pipeline qui sont deux fois plus rapides

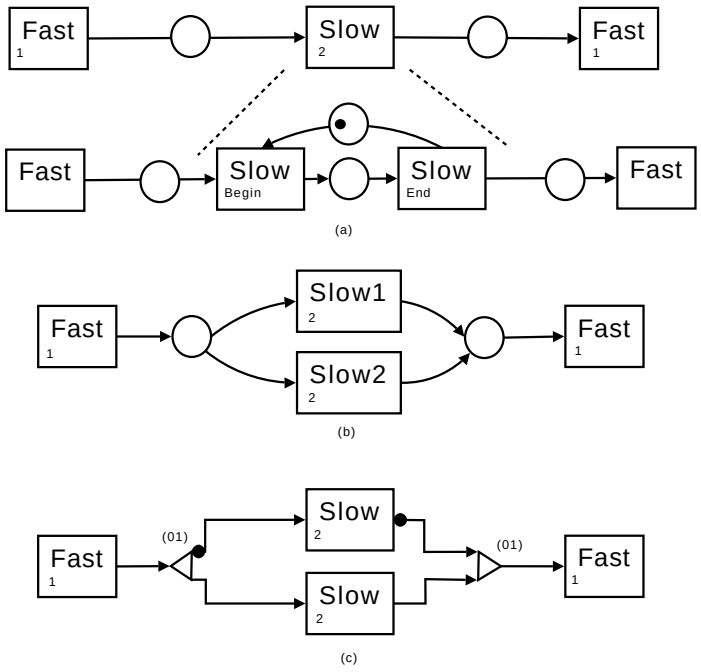


FIG. 5.1 – Exemple de pipeline - TMG vers Synchrone (a) TMG avec conflit (b) Kahn MG (c)

(moins de latence) que celui du centre. Il faudra alors 4 cycles d'horloge afin de traverser l'ensemble du pipeline et 2 cycles d'horloge entre chaque nouvelle donnée (cycle Begin/End).

Dans la figure 5.1(b) nous tentons d'améliorer la situation en terme de débit en dupliquant les éléments les plus lents afin d'amener le débit à 1 donnée produite par cycle d'horloge. Nous avons décrit un réseau de Petri avec un conflit sur une place, le système n'est plus confluent. Nous ne savons pas dans quel ordre nous allons exécuter les transitions *Slow*, la sémantique des réseaux de Petri ne précise rien à ce sujet. Nous avons introduit une forme de non déterminisme. Pourtant obtenir un système de ce type serait extrêmement intéressant car nous avons doublé le débit.

Finalement, la figure 5.1 (c) montre le type de résultat que nous souhaiterions obtenir, la place avec les conflits est transformée en un réseau de distribution avec des nœuds *select* et *merge*annotés par des ordonnancements *k*-périodiques qui sont déterministes (ordre des transitions tirées explicite), et dans cet exemple l'ordre d'arrivée des jetons est assuré. Nous appelons un tel modèle un *Kahn Marked Event Graph*. Nous avons levé le non-déterminisme introduit par le conflit. Suivons par exemple le chemin du premier jeton arrivant sur ce système : ce jeton

passe par l'étage *Fast* à gauche et arrive sur un select, puis il va être aiguillé sur la sortie 0 (boule noire) arrivant sur le composant Slow en haut, après calcul ce dernier émettra son jeton sur la branche 0 du merge jusqu'à atteindre Fast. Nous pouvons faire la même chose pour un deuxième jeton qui passera par l'étage Slow du bas. Nous observerons que les ordonnancements seront revenus dans leurs positions initiales, le premier étage Slow étant prêt à calculer.

Nous voulons maintenant étendre nos travaux sur les ordonnancements statiques *k*-périodiques en les appliquant sur un modèle plus général : une combinaison d'un Marked Event Graph et de réseaux de Kahn. Pour simplifier, nous nous restreignons à l'aspect "Kahn" de deux nœuds spécifiques *select* et *merge*, avec des conditions *k*-périodiques (reproduisant le même pattern à l'infini). Mais cette fois 0 signifie en "haut", et 1 signifie en "bas" dans la figure 5.1.

Travaux Reliés Fortement associés à notre travail, nous avons les travaux effectués par Edward Lee et ses collègues dans le cadre des graphes Synchronous Data Flow (SDF) et ses extensions [18, 61] : le plus général BDF et le domaine DDF avec le problème inhérent que l'ordonnancement statique avec des FIFOs de taille bornée devient alors indécidable. D'autres sources d'inspiration ont été, comme mentionné précédemment, la théorie N-Synchrone les formalismes synchrones multi-horloges, les théories d'ordonnancement cyclique basées sur les Timed Marked Event Graphs, ainsi que le modèle Cyclo-Static [11] qui est une extension de SDF permettant de décrire des algorithmes changeant cycliquement, mais avec un comportement prédéfini.

Plan Nous décrivons notre modèle Kahn Marked Event Graph (KMG) qui est déterministe avec des nœuds de contrôle spécifiques annotés par des nœuds N-Synchrones. Nous fournissons une interprétation des processus élémentaires des réseaux de Kahn dans notre modèle. Puis nous montrons que nous pouvons décider s'il existe un ordonnancement où les FIFOs sont de tailles finies, au travers d'une abstraction sur des graphes SDF. Nous discutons ensuite de la vivacité, de préservation d'ordre des jetons et d'une forme canonique pour comparer les KMGs.

5.2 Kahn Marked Event Graphs

Nous ajoutons aux Marked Event Graphs deux nouveaux types de nœuds : *merge* et *select*. Chaque nœud merge nécessite deux arêtes (ordonnées), ainsi qu'un mot (k-périodique) d'aiguillage, et une arête (exactement) en sortie. De son côté chaque nœud select nécessite exactement une arête et un mot d'aiguillage en entrée, et deux arêtes (exactement) en sortie.

Dans le manuscrit, nous noterons le *merge* avec un triangle pointant vers le bas, et le *select* par un triangle pointant vers le haut, comme dans les figures 5.2,5.3,5.4.

Dans l'interprétation sous-jacente, l'opérateur *merge* "consomme" les jetons sur ses second et troisième arguments en suivant son premier argument. Les trois arguments sont définis formellement par des mots binaires infinis, pour le premier les 0 et les 1 signifient "droite" et "gauche", plutôt que "présent" ou "absent" au contraire des arguments deux et trois. Le cas du *select* est similaire. Nous appelons *conditions sur flots* les mots binaires infinis associés au premier argument, et *flots de jetons* les autres.

5.2.1 Opérateurs *on* et *when* du NSynchrone

Nous allons définir des règles de manipulations (permutations, etc...) sur les nœuds merge et select. Elles nécessiteront des transformations sur les mots d'aiguillage. Pour définir ces dernières nous utilisons des opérateurs dédiés sur les mots binaires infinis k-périodiques, définis maintenant.

Définition 15 (Opérateurs "On" et "When")**.** Soit b, w, w', w_0, w_1 des mots infinis binaires dans $\mathbb{B}^{\mathbb{N}}$. Alors les opérateurs $w\,on\,b$ et $w'\,when\,w$ sont définis inductivement comme :

$$\begin{aligned} w\,on\,(\mathbb{0}.b) &= 0.(w\,on\,b) \\ \forall x \in \mathbb{B},\ (x.w)\,on\,(\mathbb{1}.b) &= x.(w\,on\,b) \\ \forall x \in \mathbb{B},\ (x.w')\,when\,(\mathbb{0}.w) &= w'\,when\,w \\ \forall x \in \mathbb{B},\ (x.w')\,when\,(\mathbb{1}.w) &= x.(w'\,when\,w) \end{aligned}$$

Une utilisation primaire de *when* arrive lorsque le premier paramètre est un sous-flot du second. Alors il est en quelque sorte le contraire de *on*, comme montré formellement dans la propriété suivante.

Propriété 4.

$$\begin{aligned} (w\,on\,\mathbb{1}^{|w|_1}) &= w \\ w\,when\,w &= 1^{|w|_1} \\ b &= (w\,on\,b)\,when\,w \\ (w' \sqsubseteq w) &\Rightarrow (w' = w\,on\,(w'\,when\,w)) \\ rate(w\,on\,b) &= rate(b).rate(w) \\ (w' \sqsubseteq w) &\Rightarrow (rate(w'\,when\,w) = rate(w')/rate(w)) \end{aligned}$$

Les équations dans la seconde ligne peuvent aussi être exprimées par $(a = c\,on\,b) \Rightarrow (b = a\,when\,c)$.

5.2.2 Opérateurs merge et select

Ces deux nouvelles sortes de nœuds : *select* (SN) et *merge* (MN) fonctionneront comme des commutateurs permettant de démultiplexer un flot en entrée dans différents nœuds en sortie (pour les SNs) et inversement de multiplexer plusieurs entrées vers une seule sortie (pour les MNs). Nous ne considèrerons que le cas binaire (*select de 2* et *merge de 2*), cela ne constitue en rien une restriction.

Définition 16 (Opérateurs "select" et "merge")**.** $merge(b, w_0, w_1)$ où $b \in [\mathbb{O}, \mathbb{1}]$ est la condition d'aiguillage et $\forall x \in [0,1]$ w_x sont des flots binaires (définis dans la section 4.1.2), est défini inductivement comme suit :

$$
\begin{aligned}
merge(\mathbb{O}.b, 0.w_0, 0.w_1) &= 0.merge(\mathbb{O}.b, w_0, w_1) \\
merge(\mathbb{O}.b, 0.w_0, 1.w_1) &= 0.merge(\mathbb{O}.b, w_0, 1.w_1) \\
merge(\mathbb{O}.b, 1.w_0, 0.w_1) &= 1.merge(b, w_0, w_1) \\
merge(\mathbb{O}.b, 1.w_0, 1.w_1) &= 1.merge(b, w_0, 1.w_1) \\
merge(\mathbb{1}.b, 0.w_0, 0.w_1) &= 0.merge(\mathbb{1}.b, w_0, w_1) \\
merge(\mathbb{1}.b, 1.w_0, 0.w_1) &= 0.merge(\mathbb{1}.b, 1.w_0, w_1) \\
merge(\mathbb{1}.b, 0.w_0, 1.w_1) &= 1.merge(b, w_0, w_1) \\
merge(\mathbb{1}.b, 1.w_0, 1.w_1) &= 1.merge(b, 1.w_0, w_1)
\end{aligned}
$$

Enfin, nous introduisons la notation pour l'opérateur *select*, afin de diviser les flux en deux, en accord avec la condition booléenne sur le flot : $select(b,w) = \langle \bar{w} on b \,|\, w on b \rangle$. Nous écrivons $select_1(b,w) = w on b$, et $select_0(b,w) = w on \bar{b}$ (l'opérateur *on* est défini dans la section 5.2.1).

Définition 17. Soit W un ensemble fini de noms de variables de flot, partionnés dans trois sous-ensembles disjoints W_{in}, W_{out}, et W_{local} ; soit aussi B un ensemble fini de conditions sur flots.

Nous appelons un *select/merge Net (SMN)* un système d'équations consistant de la forme $w = merge(b, w_0, w_1)$ ou $\langle w_0, w_1 \rangle = select(b, w)$, tel que pour chaque nom du flot dans W_{in} et W_{out} apparaissent exactement une fois sur le côté droit (utilisation) et réciproquement exactement une fois du côté gauche (définition), et chaque nom dans W_{local} apparait exactement une fois de chaque côté. De plus, nous demandons à ce que le graphe de dépendance obtenu entre les noms locaux soit acyclique.

De manière plus formelle un SMN est une structure définie comme suit :

Définition 18 (select-merge-Net)**.**
Un select-merge-Net est un n-uplet

$SMN = < \mathcal{S}, \mathcal{M}, W_{in}, W_{out}, W_{local} >$ où :
- $\mathcal{S}$ est un ensemble de nœuds *select*. Un nœud avec une entrée $w1 \in W_{in} \cup W_{local}$ et deux sorties $w2, w3 \in W_{out} \cup W_{local}$ tel que $\langle s2, s3 \rangle = select(b, s1)$, b est un mot binaire.
- $\mathcal{M}$ est un ensemble de nœuds *merge*. Un nœud avec deux entrées $w1, w2 \in W_{in} \cup W_{local}$ et une sortie $w3 \in W_{out} \cup W_{local}$ tel que $s3 = merge(b, s1, s2)$, b soit un mot binaire.
- L'ensemble des fils en entrée : $W_{in} \subset (\emptyset \times \mathcal{S}) \cup (\emptyset \times \mathcal{M})$.
- L'ensemble des fils en sortie : $W_{out} \subset (\mathcal{S} \times \emptyset) \cup (\mathcal{M} \times \emptyset)$.
- L'ensemble des fils locaux : $W_{local} \subset (\mathcal{S}' \times \mathcal{S}) \cup (\mathcal{M} \times \mathcal{S}) \cup (\mathcal{S}' \times \mathcal{M}') \cup (\mathcal{M} \times \mathcal{M}')$.
  où $X' = (X \times 0) \cup (X \times 1)$ (lorsqu'il y a deux entrées(merge)/sorties(select), on utilise 0 ou 1. Par exemple, S' signifie une des sorties d'un select).

De plus, nous rajoutons les contraintes suivantes :
- chaque paire entrée/sortie est associée à un seul et unique fil orienté : $\forall w_1, w_2 \in W, (source_1, target_1) = (source_2, target_2) \rightarrow ((source_1 = source_2) \wedge (target_1 = target_2))$.
- chaque entrée/sortie est associée à un seul et unique fil orienté : $\forall w_1, w_2 \in W w_1 \cap w_2 = \emptyset$.

Pour $w, w' \in W$ nous notons $w \rightarrow w'$ lorsque w' dépend transitivement sur w pour sa définition.

Soit C un SMN. Étant donné w et w' deux flots, i et j deux entiers, nous notons $< w, i > \rightsquigarrow_C < w', j >$ chaque fois qu'étant donné un flot w'', une des quatre situations arrive :

$$\begin{aligned} w' = merge(b, w, w'') \quad et \quad j = [\overline{b}]_i \\ w' = merge(b, w'', w) \quad et \quad j = [b]_i \\ \langle w', w'' \rangle = select(b, w) \quad et \quad i = [\overline{b}]_j \\ \langle w'', w' \rangle = select(b, w) \quad et \quad i = [b]_j \end{aligned}$$

La relation $< w, i > \rightsquigarrow_C < w', j >$ exprime exactement le fait que le i^{ieme} jeton dans w deviendra le j^{ieme} jeton dans w'. Les SMNs peuvent être composés librement tant que la propriété d'acyclicité est préservée.

Propriété 5 (Déterminisme)**.** *Étant donné n'importe quel couple $< w, i >$ tel que $w \in W_{in}, i \in \mathbb{N}$, il existe un unique couple $< w', j >$ avec $w' \in W_{out}$,*
tel que $< w, i > \rightsquigarrow_C^\star < w', j >$. Tout jeton en entrée "donne" exactement un jeton en sortie.

Démonstration. Soit b un flot de conditions infini. Soit $w_0, w_1, v \in W$. Nous considèrons deux cas :

$v = merge(w_0, w_1)$ alors nous construisons les deux relations $M_0^v = \{(i, [\bar{b}]_j) / i \in \mathbb{N}, j \in \mathbb{N}^*\}$ et $M_1^v = \{(i, [b]_j) / i \in \mathbb{N}, j \in \mathbb{N}^*\}$ (La notation $[w]_i$ a été introduite dans la définition 6 de la section 4.1.2). Alors ces deux relations sont des fonctions totales monotones. $\mathbb{N}^* \to \mathbb{N}$, avec des images disjointes et complémentaires : $M_0^v(\mathbb{N}^*) \cap M_1^v(\mathbb{N}^*) = \emptyset$ et $M_0^v(\mathbb{N}^*) \cup M_1^v(\mathbb{N}^*) = \mathbb{N}$.

$\langle w_0, w_1 \rangle = select(v)$ alors nous construisons les deux relations $S_0^v = \{([\bar{b}]_j, i) / i \in \mathbb{N}, j \in \mathbb{N}^*\}$ et $S_1^v = \{([b]_j, i) / i \in \mathbb{N}, j \in \mathbb{N}^*\}$. Alors ces relations sont des fonctions totales et monotones $\mathbb{N}^* \to \mathbb{N}$, avec des images disjointes et complémentaires ; ou, de manière équivalente S_0^v et S_1^v sont deux fonctions partielles monotones, définies sur des domaines disjoints et complémentaires, et chacunes d'elles une surjection sur $\mathbb{N}$.

À cause de ces propriétés, il est facile de prouver que, pour chaque couple $< w, i >$, $w \in W, i \in \mathbb{N}$ il y a exactement un successeur (car nos fils sont de simples liaisons point à point, et à chaque entrée/sortie un seul fil peut être connecté). avec une telle fonction dans le cas $w \notin W_{out}$. En effectuant cela par induction sur la structure du circuit, nous pouvons construire une séquence, où à cause de l'acyclicité du SMN nous atteignons un W_{out}. □

Nous pouvons maintenant définir notre notion d'équivalence

Définition 19. Soit C et D deux SMNs avec des ensembles identiques pour les noms de flots d'entrées et de sorties W_{in}, W_{out}. Nous écrivons $C \sim D$ lorsque : $\forall w \in W_{in}, \forall w' \in W_{out}, \forall i, j \in \mathbb{N}$, nous avons

$$(< w, i > \rightsquigarrow_C^\star < w', j >) \Leftrightarrow (< w, i > \rightsquigarrow_D^\star < w', j >)$$

Définition 20. Un *Kahn Marked Event Graph (KMG)* est une structure $K =< \mathcal{C}, SMN, \mathcal{E} >$ où :

- $\mathcal{C}$ est un ensemble de nœuds de calcul,
- $SMN =< \mathcal{S}, \mathcal{M}, W_{in}, W_{out}, W_{local} >$ est un select merge Net défini précédemment,
- $\mathcal{E}$ est un ensemble d'arcs avec $\mathcal{E} \subseteq (\mathcal{C} \times \mathcal{C}) \bigcup (W_{out} \times \mathcal{C}) \bigcup (\mathcal{C} \times W_{in})$,

L'intention des conditions d'aiguillage est que les décisions *purement internes* vont décrire à quel canal de sortie (entre les "points d'entrées" $\{0, 1\}$) les jetons successifs seront dirigés dans le cas du *select*, et depuis quel canal ils seront consommés à leur arrivée dans le cas du *merge*. Il doit être noté qu'alors que les nœuds select peuvent être exécutés à la vitesse des flots placés à leurs entrées, il peut arriver qu'un nœud merge soit bloqué attendant l'arrivée d'un jeton sur une entrée, alors que d'autres jetons sont arrivés de l'autre côté ; mais les décisions pour les consommations de jetons sont prises en *interne*, c'est à dire sans

prendre en compte les valeurs des données. C'est la propriété déterminante qui permet d'assurer le déterminisme et la monotonicité dans les réseaux de Kahn réguliers. Une autre manière de voir notre extension KMG aux MGs, nous ajoutons au modèle un petit réseau de contrôle avec des aiguillages déterministes.

Préservation de l'ordre des jetons

Définition 21 (Préservation de l'ordre)**.** Un SMN préserve l'ordre si pour chaque couple (w,w') $w \in W_{in}$, $w' \in W_{out}$ où $< w, i > \rightsquigarrow_c < w', k >$ et $< w, j > \rightsquigarrow_{c\prime} < w', l >$ et $i < j$, alors $k < l$. (c et $c\prime$ sont deux chemins différents.)

Nous avons posé la conjecture que sous l'hypothèse de préservation de l'ordre des jetons, un KMG fortement connexe est vivant si chaque nœud de calcul peut être exécuté au moins une fois. L'idée ici serait que le seul moyen d'obtenir un blocage depuis un KMG "sûr" où un certain nombre de jetons ont été "préservés", serait d'avoir un blocage à cause du fait que certains jetons attendent leur tour pour passer au travers d'un nœud merge, alors que d'autres sont attendus mais coincés plus haut ; ainsi s'il existe au moins un point où cela arrive, cela signifie que les jetons étaient attendus dans un ordre différent. Bien sûr cela est fort court pour une preuve, mais nous voulons étudier plus en détail ce problème. Bien sûr déterminer aussi qu'un nœud n'est exécuté qu'une seule fois peut-être aussi mauvais que de calculer l'ensemble de l'espace atteignable dans le pire des cas, mais en général il devrait être plus rapide.

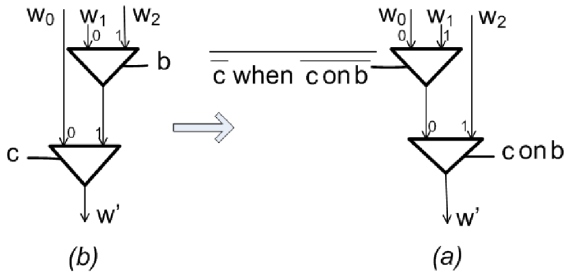


FIG. 5.2 – Permutation de merges

Propriété 6. *Les opérateurs précédents* on, when, select, *et* merge *préservent tous les propriétés* périodiques.

Dans le cas des figure 5.2 et 5.3, il est relativement facile à travers d'un calcul de déplacer de l'un (a) à l'autre (b), et aussi de (b) vers (a). Dans le cas de la figure 5.4 il est toujours possible de transformer de l'un (a) sur la gauche vers l'autre (b) ;

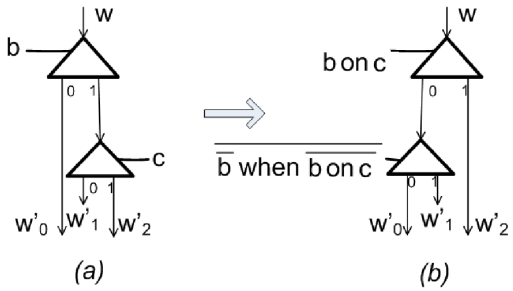


FIG. 5.3 – Permutation de selects

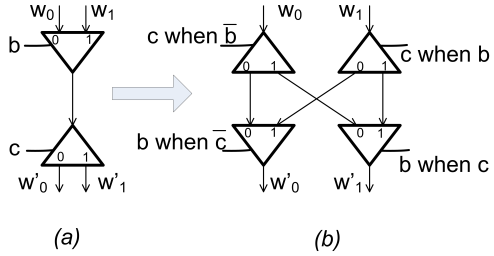


FIG. 5.4 – Permutation de merges avec selects

cependant, la transformation de celui de droite (b) vers celui de gauche (a) n'est pas toujours possible comme l'illustre la figure 5.5 à cause de l'ordre entre le jetons, car il y a un stockage temporaire d'au moins un jeton.

Propriété 7 (Identités Algébriques)**.** – *Code mort :* $merge(\mathbb{1}, w_0, w_1) = w_1$, $merge(\mathbb{0}, w_0, w_1) = w_0$,
– *Inversion ordonnancement* $\equiv$ *Inversion entrées :* $merge(\overline{b}, w_0, w_1) = merge(b, w_1, w_0)$,
– *Permutation merges :* $merge(c, w_0, merge(b, w_1, w_2)) = merge(c', merge(b', w_0, w_1), w_2)$,
où $c' = b\,on\,c$, $\overline{c} \sim \overline{b'}\,on\,\overline{c'}$, $b = c'\,when\,c$, et $\overline{b'} = \overline{c}\,when\,\overline{c'}$.

Il faut noter que les nœuds merge peuvent ralentir (et bloquer temporairement) les jetons arrivant sur un de leur flot d'entrée alors que c'est l'autre qui est attendu. De ce fait la plupart des transformations impliquant de tels nœuds ne préserveront pas la sémantique ASAP mais une version plus asynchrone. Nous devons donc définir une telle équivalence moins forte, "asynchrone", qui dit que seulement les jetons "présents" sont manipulés de manière identique dans deux expressions différentes de select/merge.

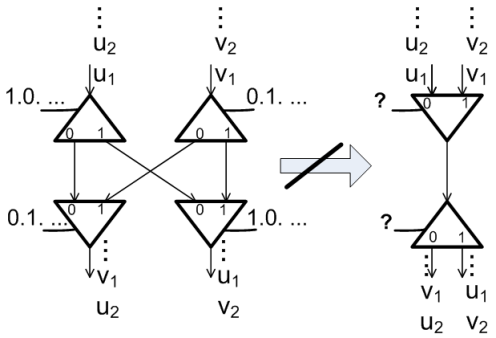


FIG. 5.5 – Permutation impossible à cause de l'ordre des jetons

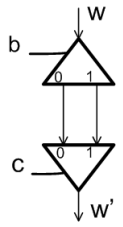


FIG. 5.6 – Simple select/merge

5.2.3 Un Exemple

Nous présentons ici une topologie d'interconnection simplifiée dans le style du processeur STI Cell avec deux anneaux contrarotatifs en utilisant notre modèle Kahn Marked Event Graph.

Dans la figure 5.7, pour faciliter la lecture nous avons représenté en gris les liaisons entre les branches $0-0$ des selects et merges et respectivement en noir les branches $1-1$. Exemple, lorsque toutes les conditions des selects et merges de l'anneau extérieur sont à 0 aucun composant n'est connecté au réseau (inversement dans l'anneau intérieur lorsqu'elles sont toutes à 1).

La figure 5.8 illustre une configuration valide possible des anneaux, où le nœud $C1$ envoie et reçoit des données de $C3$, et similairement pour $C2$ et $C4$.

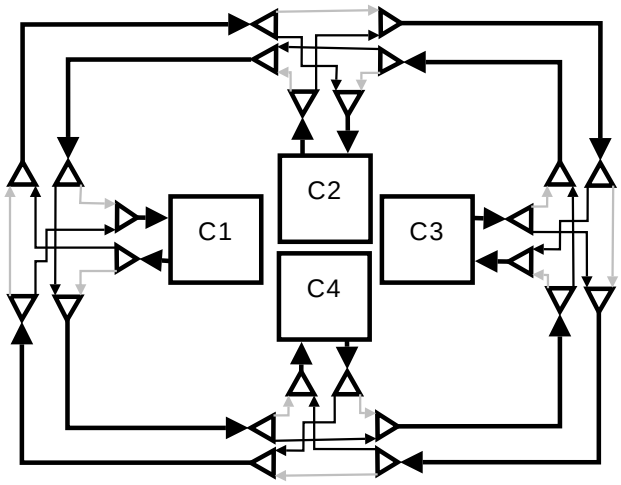


FIG. 5.7 – Exemple KMG : double anneau contrarotatif

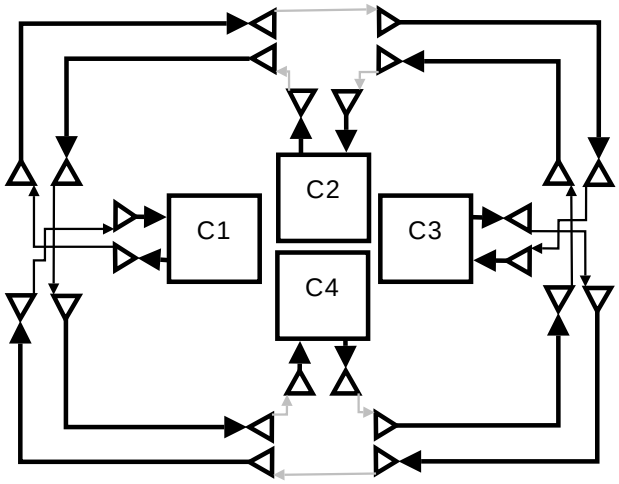


FIG. 5.8 – C1 doublement lié à C3 (et C2 → C4)

5.3 Interprétation des processus élémentaires de Kahn dans des KMGs

Pour motiver la dénomination de KMG, nous allons maintenant montrer comment interpréter dans notre formalisme une forme simple de composants locaux utilisés dans les Kahn Process Networks, en supposant une structure à états finis, une abstraction des données et des choix donc indépendants de ces données.

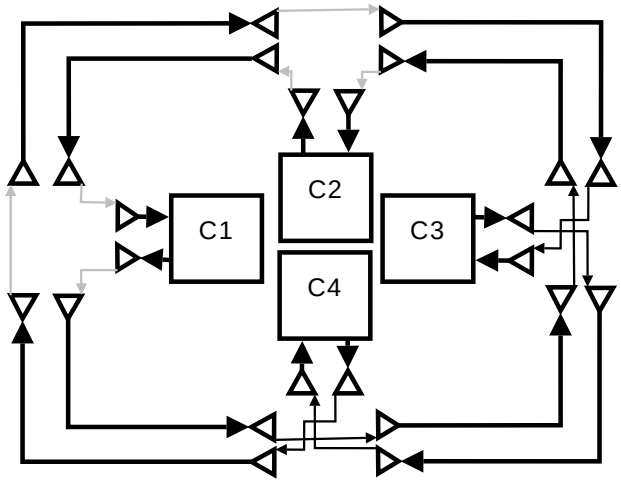


FIG. 5.9 – C1 doublement lié à C2 (et C3 → C4)

Dans le papier de référence de Kahn, ses réseaux sont des processus locaux qui communiquent avec des FIFOs de tailles non bornées. Chaque processus est séquentiel et obtenu à partir des constructions syntaxiques suivantes :

- l'affectation, la composition séquentielle, la composition $if-then-else$ et la boucle *repeat*.
- les opérateurs *wait* et *send* où *wait* est une lecture bloquante sur un canal de communication, et *send* est une écriture non bloquante agissant aussi sur un canal de communication.

Dans notre modélisation, nous abstrayons les valeurs sur les données, et supposons que les conditions de branchement de chaque $if-then-else$ sont connues et peuvent être représentées par un mot binaire k-périodique. Cela semble assez restrictif, mais il faut se rappeler que nous souhaitons calculer un ordonnancement statique.

La figure 5.10 illustre une interprétation possible du langage impératif utilisé afin de décrire les processus locaux dans notre modèle :

- (a) et (b) sont le $if-then-else$ et le *repeat*.
- (c) est la déclaration d'une entrée et d'une sortie.
- (d) et (e) décrivent le *wait* et le *send*. L'entrée *Control* représente la séquentialité de l'opération sur les deux figures. Dans le cas du *wait* il est nécessaire d'avoir un signal affirmant si la donnée issue du canal de communication est valide (*Fromchannel*).

Le programme dans la figure 5.11 (a) décrit un simple exemple et (b) sa traduction dans KMG. Partant du *Begin* englobant, nous atteignons le premier merge et select du *repeat*. Ensuite nous avons la structure du $if-then-else$ commen-

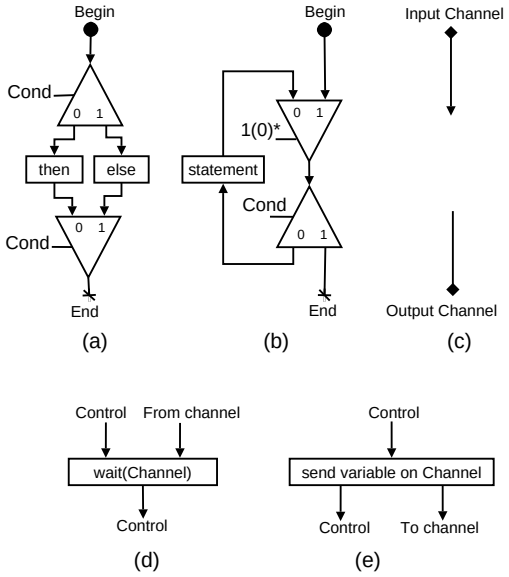
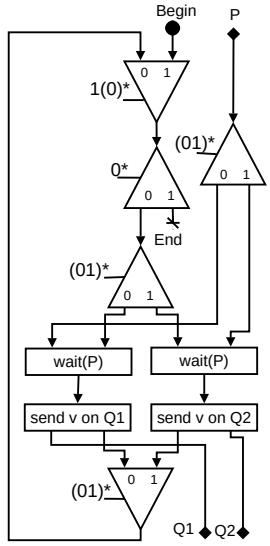


FIG. 5.10 – Interprétation (a) if-then-else (b) repeat (c) entrée/sortie (d) wait (e) send

```
Begin

integer channel A, B, C;

Process f(int in  P,
          int out Q1,
          int out Q2);

 Begin boolean i, int v;
 i:=true;
 Repeat Begin
  if i
  then  v:=wait(P),
        send v on Q1
  else  v:=wait(P),
        send v on Q2;
  i= not i;
  End
 End

Comment: Main Program;
f(A, B, C);
End
```

(a)



(b)

FIG. 5.11 – Exemple traduction Kahn (a) vers KMG (b)

çant par un merge englobant quatres nœuds et finissant par un select pour les deux branches. Le dernier select en haut à droit sur la figure 5.11 décompose le flôt de jetons arrivant de P vers les deux wait dans le $if - then - else$. Si les canaux de sorties $Q1$ et $Q2$ avaient plus d'un seul producteur, nous aurions un arbre de merge.

5.4 Abstraction de merge/select à SDF

Nous allons maintenant montrer grâce à une abstraction sur le modèle SDF que nous pouvons décider s'il existe une ordonnancement statique périodique avec des FIFOs de taille bornée sur le modèle KMG originel.

Cette construction revient à abstraire les choix d'aiguillage sur une période, pour ne retenir que le taux des jetons qui bifurquent (ou proviennent) de telle ou telle flot en sortie des nœuds select (ou en entrée des nœuds merge).

Chaque transition est transformée en une transition dans SDF avec le même nombre d'entrées/sorties et un poids de 1 pour chacune des entrées/sorties.

Un nœud merge avec le paramètre b peut être converti en un nœud SDF avec 2 entrées et 1 sortie avec des poids respectivement $|b|_1, |b|_0$ et $|b|$ associés à la première et seconde entrée et la sortie.

Un nœud select avec le paramètre b est changé en une transition SDF avec 1 entrée et 2 sorties avec des poids respectivement de $|b|$, $|b|_1$ et $|b|_0$.

Propriété 8. *Soit G un Kahn Marked Event Graph, si le graphe SDF abstrait de G est* sûr *alors G est* sûr.

Démonstration. Les nœuds originaux de G et ceux qui ont été abstraits dans le graphe SDF ont le même comportement. Les transitions sont toujours de la même nature et consomment/produisent un jeton sur toutes leurs entrées/sorties à chaque fois qu'elles sont activées. Concernant les nœuds select, nous envoyons toujours les jetons $|b|_0$ sur la sortie gauche et les jetons $|b|_1$ sur la sortie droite. L'abstraction en SDF a perdu cette information à propos de quel jeton va à droite ou à gauche, mais si nous considèrons les b jetons, le rapport est préservé sur une période. Il en va de même pour les nœuds merge.

Si le système SDF est dit *sûr*, cela signifie que l'ensemble des différents états du système est fini. Ainsi il est aussi fini pour G. Comme un état du système est défini par la distribution des jetons, cela signifie que le nombre de jetons pour chaque place est alors fini. Il existe donc un $k \in \mathbb{N}$ tel que G soit *k-sûr*. □

La figure 5.12 illustre un exemple d'une abstraction d'un Kahn Marked Event Graph (a) à un graphe SDF (b).

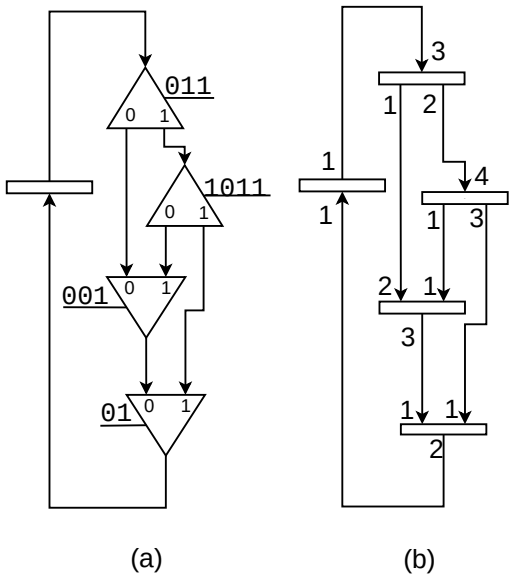


FIG. 5.12 – Conversion d'un (a) KMG à (b) SDF

Vivacité

Une fois que nous disposons d'un critère pour établir la sûreté du système (c'est à dire que toutes les places ont une taille finie), nous savons que l'espace d'états atteignables est aussi fini. Ainsi, en explorant cet espace d'états nous pouvons vérifier l'existence d'interblocage, de famine et généralement considérer le problème de vivacité d'une manière décidable. Il peut être remarqué ici que, sous l'hypothèse ASAP, il y a un successeur (au plus) pour chaque état. Mais cela pose le problème de l'efficacité de la représentation des états ; ils contiennent maintenant à la fois les marquages de jetons et les indices indiquant quelle profondeur d'exécution a été effectuée pour chaque flot de conditions booléennes pour les opérateurs select et merge. De plus, la construction de l'espace d'états atteignables permet aussi de calculer exactement les délais et les tailles des places nécessaires entre les différentes routes que les jetons peuvent emprunter pour atteindre les mêmes nœuds de calcul. Cela est fortement relié aux précédents résultats d'ordonnancement cycliques sur les systèmes insensibles à la latence. En fait, il peut être facilement vu, qu'à cause du fait que chaque état dispose d'un successeur unique dans la sémantique ASAP et que l'espace d'état est fini alors dans le cadre de notre modèle étendu avec les nœuds de contrôle merge/select resteront ultimement k-périodiques. Où, et combien, de cellules il faut pour les places peut être déduit de ces ordonnancements.

5.5 Forme normale et transformation des canaux

Les identités remarquables sur les réseaux de nœuds merge/select permettent de conduire des transformations. Celles-ci amènent à des pliages/dépliages de liens et de connexions entre les nœuds de calcul, qui peuvent ainsi s'adapter à une topologie éventuelle de canaux de communications. Nous pouvons effectuer des transformations afin d'optimiser l'utilisation de ces ressources, sachant que plus on partage des liens, plus l'ordonnancement et l'aiguillage des jetons/valeurs devient délicat ; alors que la multiplication des "tuyaux" autorise bien entendu de plus grands débits de jetons (c'est l'essence des transformations des figures 5.5 et 5.4). Nous montrons ici comment tout réseau SMN peut s'expanser de manière canonique dans une forme où aucun lien n'est partagé pour transporter des jetons/valeurs avec des extrémités différentes aux deux bouts. Ces réseaux contiennent des composants de type Simple select merge *locaux* (illustré dans la figure 5.6) pour autoriser le déséquencement sur des liens non partagés (par exemple au bas de la figure 5.13).

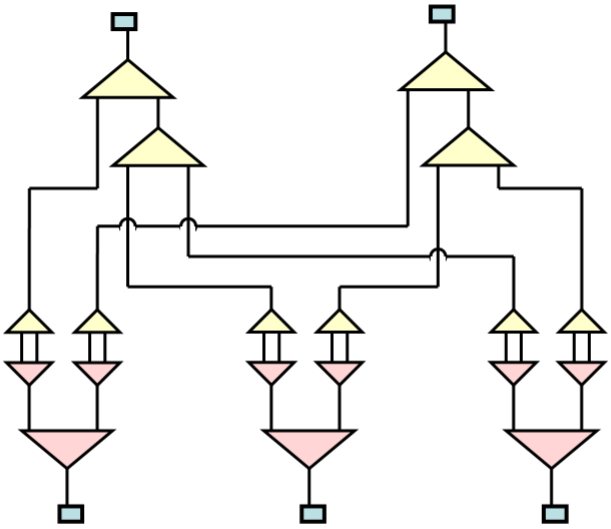


FIG. 5.13 – Forme normale SMN

5.6 Résumé

Dans ce chapitre nous avons introduit un modèle sans conflit et déterministe mêlant à la fois les réseaux de *Kahn* et les *Marked Event Graphs*. Ce modèle débute d'un Marked Event Graph où nous avons ajouté deux nouveaux nœuds : *merge* et *select* annotés avec des mots binaires spécifiques *N-Synchrones*. Le contrôle dans ce cadre est indépendant des données. Nous appelons une telle structure un *Kahn Marked Event Graph* (KMG).

Nous montrons qu'un tel réseau est *sûr* (places de taille bornée) au travers d'une abstraction sur les modèles SDF : l'abstraction transforme chaque nœud *select/merge* par un nœud SDF où chaque entrée/sortie est annotée par le nombre d'occurrences de 1 et de 0 respectivement dans le mot *N-Synchrone* associé. Nous utilisons le théorème majeur du modèle SDF qui permet de déterminer si le système admet un ordonnancement avec des FIFOs de taille bornée.

La condition de *sûreté* permet d'établir si le KMG a un espace d'états finis ; alors nous pouvons appliquer une simulation ou des techniques issues du model-checking pour vérifier la vivacité du système (absence de famine et d'interblocage) dans le cas de système avec au moins une partie fortement connexe. Nous pouvons aussi déterminer pour chaque transition son ordonnancement *k-periodique* et les tailles des places nécessaires.

Chapitre 6

Conclusion

Dans cette thèse, nous avons mis en exergue des liens existants entre les modèles Synchrones et Asynchrones pour une modélisation dite Latency Insensitive qui revient à désynchroniser puis resynchroniser (en fonction des latences) la spécification initiale du système. Notre approche est centrée autour du modèle des Marked Event Graphs avec une sémantique ASAP. Ces modèles ont la particularité d'être sans choix et confluents (préservation des potentialités de franchissement des transitions). Ils partagent des hypothèses de correction semblables telles que la vivacité et la sûreté. La sûreté dans ce contexte signifie que la taille de toute place est bornée. Ils acceptent des ordonnancements cycliques réguliers (dits k-périodiques) sur lesquelles on peut effectivement calculer et réguler les débits de données.

Nous avons pu effectuer un lien formel grâce à une équivalence de flot entre le synchrone et le Latency Insensitive, et entre le synchrone et les Marked Event Graphs. Cette équivalence de flot affirme que nous avons des ordres partiels compatibles sur les évènements par rapport à la spécification synchrone. Nous avons montré que le modèle Latency Insensitive est une instance particulière des Marked Event Graphs avec des places dont la capacité est égale à 2.

Nous avons décrit ensuite une implémentation synchrone et formelle des Shells et Relay-Stations qui sont les blocs de base du modèle Latency Insensitive. Ce travail a permi de donner une explication opérationnelle du fonctionnement d'un modèle Latency Insensitive en détaillant ces Stations de Relais et ces Shells. Cette implémentation a été validée par un ensemble de propriétés qui ont été décrites et vérifiées formellement grâce au langage synchrone Esterel et un model-checker. Sommairement, la Station de Relais fournit deux fonctions : elle permet de répéter le signal qui ne peut se propager en un cycle d'horloge d'une part, d'autre part

elle implante une portion du protocole de contrôle de flot assurant la conservation des données en cas de congestion, ainsi que l'ordre des évènements. Le Shell implante aussi deux fonctions : une partie du protocole de contrôle de flot, et il commande aussi l'horloge du module synchrone associé. L'horloge du Shell est activée lorsque tous les signaux d'entrées sont présents et qu'il y a suffisamment de place sur toutes les Stations de Relais présentes sur les sorties.

Nous avons ensuite exploité le fait que le modèle Latency Insensitive est un cas particulier de Marked Event Graphs, en utilisant des résultats existants sur l'ordonnancement statique de parties fortement connexes. Nous avons introduit une technique d'ordonnancement statique dénommée *égalisation*. L'égalisation "ralentit" les chemins du système qui sont trop rapides, tout en conservant le débit global du système originel. Cette technique permet de déterminer les chemins de calculs où nous pouvons rajouter un nombre d'étages dans certains pipelines sans altérer le débit. Ceci autorise par exemple à utiliser moins de surface et/ou moins d'énergie. L'algorithme d'égalisation se décrit de la manière suivante : nous recherchons la liste des cycles du système à étudier et calculons le débit optimal du système, puis nous construisons un système d'équations résolu par un solveur de programmation linéaire en nombre entier. Ce dernier va modifier la topologie de communication de notre système en rajoutant des latences supplémentaires tout en conservant le même débit global. Nous déterminons ensuite les endroits où il reste des résidus fractionnaires. Finalement, nous ordonnançons le système en utilisant une simulation jusqu'à atteindre le point fixe, c'est à dire lorsque l'ordonnancement statique devient périodique. Cette simulation va générer les ordonnancements à la fois pour les instants d'activation des nœuds de calcul mais aussi pour les registres fractionnaires chargés de "gommer" les résidus fractionnaires.

Puis, nous avons introduit dans ces précédents modèles une notion de contrôle restreint. Ce contrôle permet d'assurer à la fois déterminisme et sûreté. Nous introduisons dans le modèle deux nœuds additionnels nommés *Select* et *Merge* à la manière de dé-multiplexeur et multiplexeur. Ces nœuds portent en guise de condition un pattern ultimement k-périodique. Nous avons montré grâce à une abstraction effectuée sur le modèle SDF (Synchronous Data Flow) que nous pouvions vérifier s'il existe un ordonnancement statique avec buffers de taille bornée. Ce modèle est déterministe car il partage la même hypothèse que les réseaux de Kahn : aucune décision de l'ordonnancement ne dépend de la valeur d'une entrée.

Ces résultats pourraient être prolongés par des travaux futurs, dont nous allons maintenant mentionner quelques pistes. Nous les avons décomposées en différentes catégories : modélisation formelle, implantation de LID, ordonnancement statique et le contrôle.

Autres axes de recherche

Modélisation formelle Jusqu'ici nous n'avons considéré que des modèles mono-horloge. Nous souhaitons réintroduire les notions de "sampling" issues de SDF, afin de décrire des systèmes plus complexes et ensuite étendre ce modèle vers du multi-horloge. Un problème important dans SDF est comment dimensionner (et minimiser) la taille des FIFOs interconnectant les transitions [58].

Le modèle LID suppose que nous pouvons arrêter dans l'instant chaque composant (en utilisant du Clock-Gating), pour des systèmes suffisamment large cette hypothèse est remise en cause. Cela remettra en question les bornes sur les tailles des buffers d'une part, et d'autre part l'ensemble du protocole de contrôle de flot.

Il serait aussi intéressant de modifier l'hypothèse que les communications sont des fils et introduire un intervalle à la place par exemple. Ceci permettrait de modéliser une classe plus large et moins rigidemment prévisible d'applications.

Implémentation de LID Les travaux de Cortadella, Kishinevsky *et al.* [50] sur les registres "élastiques" montrent que la réalisation pratique des notions de stations de relais et de shells peut encore être améliorée.

Égalisation Durant l'initialisation, le débit peut être beaucoup plus lent/rapide et les ressources de stockage temporaires peuvent n'être utilisées que lors de l'initialisation seulement, et pas dans la phase périodique de l'ordonnancement statique. Des initialisations plus courtes et/ou plus équilibrées du point de vue du débit sont nécessaires afin de minimiser ces ressources de stockage temporaires. Il est aussi possible d'utiliser des ordonnancements plus équilibrés afin de minimiser le nombre de ressources de stockage temporaires utilisées.

Notons que notre technique d'"égalisation" permet de contraindre la "forme" de l'ordonnancement statique obtenu ; l'"idéal" étant que l'ordonnancement périodique obtenu pour un nœud soit celui de son prédécesseur modulo la latence existante entre ces nœuds (modulo au sens "avec n rotations", où n est la latence du fil). Ceci permettrait alors de factoriser l'ensemble de ces ordonnancements périodiques et d'en distribuer un seul de la même manière que nous distribuons une horloge. Cette factorisation permettrait d'obtenir des gains importants en surface dans le cas d'une implantation.

Nous pouvons étudier différents critères parmi les ordonnancements admissibles, comme d'éviter les pics de consommation (quand un maximum de nœuds travaillent à certains instants, alors que d'autres instants ne servent qu'à propager les données sur les fils), ou faire fonctionner les nœuds de calcul en "burst mode" (activations successives sans interruption aussi longtemps que possible).

On peut aussi étudier l'utilisation des latences virtuelles pour reconcevoir certains composants afin de limiter la consommation.

Contrôle La vision du contrôle que nous avons est restrictive du point de vue de la puissance d'expression. Nous souhaiterions étendre notre notion de contrôle vers une classe plus large, tout en conservant la possibilité d'assurer que les buffers soient de taille bornée.

Table des figures

Bibliographie

[1] Tochéou Amagbegnon, Loïc Besnard, and Paul Le Guernic. Implementation of the Data-flow synchronous language Signal. In *Proceedings PLDI'95*, 1995.

[2] François Anceau. A synchronous approach for clocking vlsi systems. *IEEE Journal of Solid-State Circuits*, 17(1) :51–56, February 1982.

[3] Charles André. Use of the behaviour equivalence in place-transition net analysis. In *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*, pages 241–250, 1981.

[4] Charles André. Representation and Analysis of Reactive Behaviors : A Synchronous Approach. In *Computational Engineering in Systems Applications*, pages 19–29, 1996.

[5] Charles André. Semantics of S.S.M. Technical report, I3S, CNRS, Esterel Technologies, 2003.

[6] Charles André. Comparaison des styles de programmation de langages synchrones. Technical report, CNRS - UNSA - INRIA, Routes des Lucioles, Sophia Antipolis, France, Juin 2005.

[7] Semiconductor Industry Association. The international technology roadmap for semiconductors, 2005 edition. Technical report, SEMATECH :Austin, TX, 2005.

[8] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and Linearity*. Wiley, 1992.

[9] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Programming with events and relations : the signal language and its semantics. *Science of Computer Programming*, 16 :103–149, 1991.

[10] Gérard Berry and Georges Gonthier. The Esterel Synchronous Programming Language : Design, Semantics, Implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[11] Greet Bilsen, Marc Engels, Rudy Lauwereins, and Jean Peperstraete. Cyclo-static dataflow. In *IEEE Transactions on Signal Processing*, volume 4, February 1996.

[12] Amar Bouali. Xeve, an ESTEREL Verification Environment. In *CAV '98 : Proceedings of the 10th International Conference on Computer Aided Verification*, pages 500–504, London, UK, 1998. Springer-Verlag.

[13] Julien Boucaron, Robert de Simone, and Jean-Vivien Millo. Formal methods for scheduling of latency-insensitive designs. *EURASIP Journal on Embedded Systems*, 2007 :Article ID 39161, 16 pages, 2007. doi :10.1155/2007/39161.

[14] Julien Boucaron, Jean-Vivien Millo, and Robert de Simone. Another glance at relay stations in latency-insensitive design s. In *FMGALS'05*, 2005.

[15] Julien Boucaron, Jean-Vivien Millo, and Robert de Simone. Latency insensitive design : Dynamic and static scheduling with proper formal devices. In *SAME 2006 Proceedings*, 2006.

[16] Julien Boucaron, Jean-Vivien Millo, and Robert De Simone. Latency-insensitive design and central repetitive scheduling. In *Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proc eedings. Fourth ACM and IEEE International Conference on*, pages 175– 183, Piscataway, NJ, USA, 2006. IEEE Press.

[17] Frédéric Boussinot and Robert de Simone. The esterel language. *Proc. IEEE*, 79 :1293–1304, September 1991.

[18] Joseph T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using t he Token Flow Model*. PhD thesis, University of California, Berkeley, CA 94720, 1993.

[19] Jacques Carlier and Phillipe Chrétienne. *Problème d'ordonnancement : modélisation, complexité, algorithmes*. Masson, Paris, 1988.

[20] Luca P. Carloni. The role of back-pressure in implementing latency-insensitive systems. In *FMGALS 2005 Proceedings*, pages 81–100, Verona, July 2005. Columbia University NY, ENTCS.

[21] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. Latency insensitive protocols. In N. Halbwachs and LNCS 1633 D. Peled, editors, *Proc. of the 11th Intl. Conf. on Computer-Aided Verification (CAV)*, page 12. UC Berkeley, Cadence Design Laboratories, July 1999.

[22] Luca P. Carloni and Alberto Sangiovanni-Vincentelli. Combining Retiming and Recycling to Optimize the Performance of Synchronous Circuit. In *The Proceedings of the 16th Symposium on Integrated Circuits and System Design*, 2003.

[23] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Performance analysis and optimization of latency insensitive systems. In *Design Automation Conference*, pages 361–367, 2000.

[24] Luca P. Carloni and Alberto L. Sangiovanni-Vincentelli. Coping with Latency in SoC Design. *IEEE Micro*, 22(5) :24–35, September/October 2002.

[25] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, Pensylvania, May 1996.

[26] Paul Caspi and Marc Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Université Pierre et Marie Curie, Laboratoire LIP6, 2000.

[27] Mario R. Casu and Luca Macchiarulo. A Detailed Implementation of Latency Insensitive Protocols. In *FMGALS 2003 Proceedings*, 2003.

[28] Mario R. Casu and Luca Macchiarulo. A New Approach to Latency Insensitive Design. In *DAC'2004*, 2004.

[29] Mario R. Casu and Luca Macchiarulo. Issues in implementing latency insensitive protocols. In *Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[30] Ajanta Chakraborty and Mark R. Greenstreet. A Minimalist Source-Synchronous Interface. In *Proceedings of the 15th IEEE ASIC/SOC Conference*, pages 443–447, September 2002.

[31] Daniel M. Chapiro. *Globally-asynchronous locally synchronous systems*. PhD thesis, Stanford University, October 1984.

[32] Tiberiu Chelcea and Steven M. Nowick. Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols. In *Design Automation Conference*, pages 21–26, 2001.

[33] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau, and Marc Pouzet. N-synchronous kahn networks. In *POPL 2006 Proceedings*, January 2006.

[34] F. Commoner, Anatol W.Holt, Shimon Even, and Amir Pnueli. Marked Directed Graph. *Journal of Computer and System Sciences*, 5 :511–523, october 1971.

[35] Jordi Cortadella, Michael Kishinevsky, and Bill Grundmann. Synthesis of synchronous elastic architectures. In *DAC*, pages 657–662, 2006.

[36] Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems*, 9(4) :385–418, October 2004.

[37] Vincent H. Van Dongen, Guang R. Gao, and Qi Ning. A polynomial time method for optimal software pipelining. In *Proceedings of the Second Joint International Conference on Vector and Parallel Processing : Parallel Processing*, volume 634 of *LNCS*, pages p613–624, 1992.

[38] François-Xavier Fornari. *Optimisation du contrôle et implantation en circuits de programmes ESTEREL*. PhD thesis, École des mines de Paris, 1995.

[39] Even G., Naor J., Schieber B., and M. Sudan. Approximating minimum feedback sets and multi-cuts in directed graphs. In *4th Int. Conf. on Integer Prog. and Combinatorial Optimization*, number 920 in Lecture Notes in Computer Sciences, pages 14–28. Springer-Verlag, 1995.

[40] H. Genrich. Das zollstationenproblem. Technical Report GMD-I5/69-01-15, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, 1969.

[41] Steve Golson. The human eco compiler. In *Synopsys Users Group Conference*, Boston, 2004.

[42] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Didier Pilaud. The synchronous data flow programming language lustre. *Proc. IEEE*, 79 :1305–1320, September 1991.

[43] David Harel. Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, 8(3) :231–274, June 1987.

[44] Kathryn Heninger, David L. Parnas, John E. Shore, and John W. Kallander. Software requirements for the a-7e aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[45] Kathryn L. Heninger. Specifying software requirements for complex systems : New techniques and their application. *IEEE Transactions on Software Eng.*, SE-6(1) :2–13, January 1980.

[46] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, 1978.

[47] Gilles Kahn. The semantics of a simple language for parallel programming. In *Inform. Process. 74 : Proc. IFIP Congr. 74*, pages 471–475, 1974.

[48] Michael Kishinevsky, Jordi Cortadella, Bill Grundmann, Sava Krstic, and John O'Leary. Synchronous elastic circuits. In *CSR*, pages 3–5, 2006.

[49] Ivan S. Kourtev and Eby G. Friedman. *The VLSI Handbook*, chapter 47 - System Timing, pages 47–1 – 47–32. IEEE Press/CRC Press LLC, 1999.

[50] Sava Krstic, Jordi Cortadella, Michael Kishinevsky, and John O'Leary. Synchronous elastic networks. In *FMCAD*, pages 19–30, 2006.

[51] Eugene Lawler. *Combinatorial Optimization : Network and Matroids*. Holt, Rinehart and Winston, 1976.

[52] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE transactions on computers*, C-36(1) :24–35, 1987.

[53] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proceeding of the IEEE*, 75(9) :1235–1245, 1987.

[54] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design*, 17(12) :1217–1229, December 1998.

[55] Charles E. Leiserson and James B. Saxe. Retiming Synchronous Circuits. *Algorithmica*, 6, 1991.

[56] Florence Maraninchi. *Document d'Habilitation à Diriger des Recherches : Modélisation et validation des systèmes réactifs : un langage synchrone à base d'automates*. PhD thesis, Université Joseph Fourier (Grenoble I), 22 mai 1997.

[57] Florence Maraninchi and Yann Rémond. Argos : an automaton-based synchronous language. *Computer Languages*, (27) :61–92, 2001.

[58] Olivier Marchetti. *Dimensionnement des mémoires pour systèmes embarqués*. PhD thesis, Université Pierre et Marie Curie, Paris, France, December 2006.

[59] Douglas Matzke. Will physical scalability sabotage performance gains ? *IEEE Computer*, 8(9) :37–39, September 1997.

[60] Frédéric Mignard. *Compilation du langage ESTEREL en systèmes d'équations booléennes*. PhD thesis, École des mines de Paris, 1994.

[61] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California, Berkeley, CA 94720, December 1995.

[62] Luca P.Carloni, Kenneth L.McMillan, and Alberto L.Sangiovanni-Vincentelli. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2001.

[63] Luca P.Carloni, Kenneth L.McMillan, Alexander Saldanha, and Alberto L.Sangiovanni-Vincentelli. A Methodology for Correct-by-Construction Latency Insensitive Design. In *THE BEST OF ICAD*, 1999.

[64] Dumitri Potop-Butucaru. *Optimizations for faster simulation of Esterel programs*. PhD thesis, École des Mines de Paris, France, November 2002.

[65] Chander Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Net s*. PhD thesis, MIT, Cambridge, Massachusetts, USA, September 1973.

[66] François R.Boyer, El Mostapha Aboulhamid, Yvon Savaria, and Michel Boyer. Optimal Design of Synchronous Circuits Using Software Pipelining. In *Proceedings of the ICCD'98*, 1998.

[67] François R.Boyer, El Mostapha Aboulhamid, Yvon Savaria, and Michel Boyer. Optimal design of synchronous circuits using software pipelining techniques. In *ACM Transactions on Design Automation of Electronic Systems*, volume 6, pages 516–532, October 2001.

[68] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4) :590–599, October 1968.

[69] Valérie Roy and Robert de Simone. Auto/autograph. In *CAV '90 : Proceedings of the 2nd International Workshop on Computer Aided Verification*, pages 65–75, London, UK, 1991. Springer-Verlag.

[70] Cheng Sheng. Synchronous latency insensitive design in fpga. Master's thesis, Linköping University, February 2005.

[71] Syed Suhaib, Deepak Mathaikutty, David Berner, and Sandeep Shukla. Validating families of latency insensitive protocols. *IEEE Transactions on Computers (TCOMP), Special Issue on Simulation-Based Validation*, 55(11) :1391–1401, 2006 2006.

[72] Syed Suhaib, Deepak Mathaikutty, Sandeep Shukla, David Berner, and Jean-Pierre Talpin. A functional programming framework for latency insensitive protocol validation. In *Proceedings of the Second Workshop on Globally Asynchronous, Locally Synchronous Design (FMGALS 2005)*, volume 146, pages 169–188. Electronic Notes in Theoretical Computer Science, January 2006.

[73] Christer Svensson. Synchronous Latency Insensitive Design. In *ASYNC'04*, 2004.

[74] D. Sylvester and K. Keutzer. Getting to the bottom of deep submicron. In *Proc. Intl. Conf. on Computer-Aided Design*, November 1998.

[75] Olivier Tardieu. *Loops in Esterel : from operational semantics to formally specified compilers*. PhD thesis, École des mines de Paris, 2004.

[76] Horia Alexandru Toma. *Analyse constructive et optimisation séquentielle des circuits générés à partir du langage synchrone réactif ESTEREL*. PhD thesis, École des mines de Paris, Septembre 1997.

Index

Glossaire

Clock Gating Clock gating est une des techniques utilisée pour diminuer l'énergie, elle est utilisée dans les circuits synchrones. Pour diminuer l'énergie, le clock gating utilise de la logique additionnelle pour arrêter l'arbre d'horloge, désactivant ainsi des portions du circuit de façon que les bascules (registres) ne changent pas d'état, 55

dataflow Un ordinateur dataflow, flot de données, décrit une architecture où les données sont des entités actives qui traversent le programme de manière asynchrone, contrairement à l'architecture classique von Neumann où elles attendent passivement en mémoire pendant que le programme est exécuté séquentiellement suivant le contenu du pointeur de programme (PC). On parle aussi d'ordinateur cadencé par les données, 33

ECP Engineering Change Orders, est une modification effectuée automatiquement à une représentation d'un design (RTL, netlist, layout), 122

LID

Latency Insensitive Design est une technique permettant de dériver une implantation synchrone acceptant des latences données à partir d'une spécification synchrone ne pouvant atteindre le Timing Closure à cause des latences présentes sur les fils de communication. L'idée est d'encapsuler chaque module avec un Shell et de segmenter les fils de communication trop longs en latence avec des Stations de Relais. Les Shells et Stations de Relais implantent un protocole de contrôle de flot permettant d'assurer que l'ordre partiel des évèments générés par le design insensible aux latences est compatible avec celui du modèle synchrone originel, 136

Problème Central Répétitif

Ordonnancer en une durée minimale un ensemble de tâches soumises à des contraintes temporelles de type inégalité de potentiel (date de début et fin de tâche) ; on cherche à déterminer un ensemble de dates positives rendant minimum la durée de l'ordonnancement et satisfaisant les inégalités de potentiel cf p31 de [19], 69

Software Pipelining

Le Software Pipelining est une technique utilisée pour optimiser les boucles, de façon à paralléliser ces dernières. Le Software pipelining est une exécution de type out-of-order, à l'exception faite que le réordonnancement est effectué par le compilateur (où dans le cas d'assembleur écrit à la main, par le programmeur) à la place du processeur. Quelques architectures disposent d'un support explicite pour le Software Pipelining, notamment l'architecture d'Intel IA-64, 63

Timing Closure

Timing Closure est le processus par lequel un FPGA ou un design VLSI avec une représentation physique est modifié afin d'atteindre les contraintes temporelles. La plupart des modifications sont effectuées par les outils de synthèse (EDA) basé sur des directives fournies par un designer. Le terme est quelquefois utilisé comme une caractéristique, qui est attribuée à un outil de synthèse, lorsque ce dernier fournis la plupart des caractéristiques nécessaire dans un processus de fabrication donné. Le timing closure devient plus important avec les techonologies de fabrication submicroniques, de plus en plus d'étapes du flot de fabrication doivent prendre en compte cette notion de temps. Auparavant seulement l'étape de synthèse logique devait satisfaire ces contraintes temporelles, 62

VLIW

VLIW, initiales de Very Long Instruction Word en anglais, dénote une famille d'ordinateurs dotés d'un processeur à mot d'instruction très long (couramment supérieur à 128 bits), 63

Annexe A

Latency Insensitive Design

Le paradigme synchrone est le standard de facto dans l'industrie des semi-conducteurs. Il permet de disposer d'une vue exacte sur les évènements en terme d'ordre temporel, et ainsi de prédire ses performances avec une bonne accuité. Une difficulté d'implantation notoire de ce paradigme est la propagation de l'horloge et des données vers tous les éléments de calcul. La solution courante est de distribuer l'horloge avec un arbre équilibré permettant de limiter le *skew* (horloge arrivant à différents instants sur différents composants). Néanmoins cela ne résoud pas pour autant le problème des latences au niveau des données, à cause des variations de délais sur les fils porteurs de ces dernières. Ces variations peuvent être tellement importantes que le système fonctionne à une cadence excessivement lente par rapport aux contraintes introduites lors de sa spécification.

Le but de ce chapitre est d'introduire la théorie des systèmes insensibles à la latence : partant d'une spécification synchrone qui ne peut atteindre la vitesse à cause de fils globaux trop longs, nous obtiendrons après transformation un système fonctionnant à une cadence beaucoup plus grande tout en fournissant le même comportement modulo une déformation temporelle : au lieu d'avoir le résultat en un cycle d'horloge il en faudra un nombre arbitraire potentiellement variable. Cette théorie est mise en œuvre grâce à des éléments de synchronisation additionels encapsulant chaque composant synchrone (IP, appelé *Perle*) que nous appellerons *Shells* et des répéteurs particuliers dénommés *Stations de Relais* qui seront rajoutées à chaque fois que cela s'avère nécessaire sur les fils de données afin de satisfaire la contrainte de cadence.

A.1 Théorie et état de l'art

Avant de rentrer dans le vif du sujet nous allons rappeler de manière assez détaillée d'où sont issus ces délais sur les fils et pourquoi malheureusement cette grandeur devient de plus en plus importante dans les circuits/systèmes nécessitant des performances très élevées.

Problème de délais sur les fils

Le problème de délais sur les fils est un problème connu depuis longtemps, répertorié par François Anceau dans [2]. Ces délais sur les fils sont fonction de la longueur du fil L et d'un paramètre de taille que nous notons s tel que $Wire_{Delay} \sim L^2/s^2$ et similairement le délais d'une porte logique est fonction principalement de sa taille modulo un coefficient α : $Gate_{Delay} \sim s^{\alpha}$ où $\alpha \in [1;2]$. Malgré la présentation de ce problème il y a presque 25 ans, nous pouvons toujours construire des systèmes synchrones fonctionnant à plusieurs gigahertz mais avec des problèmes considérables. Les ASICs disposant d'horloges fonctionnant à des fréquences moins élevées sont moins critiques et pourtant ils ont aussi désormais de sérieux problèmes pour parvenir au *timing-closure* (processus de prendre en compte les temps de propagation des signaux sur les fils globaux dans un circuit afin de vérifier que ce circuit fonctionne comme attendu). Si nous souhaitons faire fonctionner un système synchrone à plusieurs gigahertz cela nécessite l'utilisation d'un flot de conception en *full custom design*, c'est à dire que la majorité des tâches du flot sont alors semi-automatiques ou manuellement optimisées avec des coûts très élevés en complexité, temps de conception et de vérification.

Malgré des progrès techniques portant sur l'augmentation du nombre de couches métalliques, l'introduction du cuivre et de diélectriques plus performants, le délais moyen d'une ligne métallique avec une longueur constante (courte, ici) croît énormément avec les nouveaux procédés de fabrication. Le débit maximum obtensible est une fonction paramètre de la surface de la section du fil et de sa longueur au carré. Le facteur critique est principalement la longueur, car malheureusement augmenter la section du fil (qui revient à diminuer sa résistance) ne peut aider que lorsque nous disposons de suffisamment de surface, mais induit aussi en général une augmentation de la capacité du fil (et donc à une augmentation du délais). Tout ceci est l'affaire de savants compromis entre ces deux grandeurs et d'un nombre de paramètres sans cesse croissants dont nous n'avons pas forcément la maîtrise : par exemple la géométrie du fil, espacement avec les fils adjacents, imperfections causées par le processus de fabrication.

La solution communément utilisée afin de parvenir à augmenter le débit de la

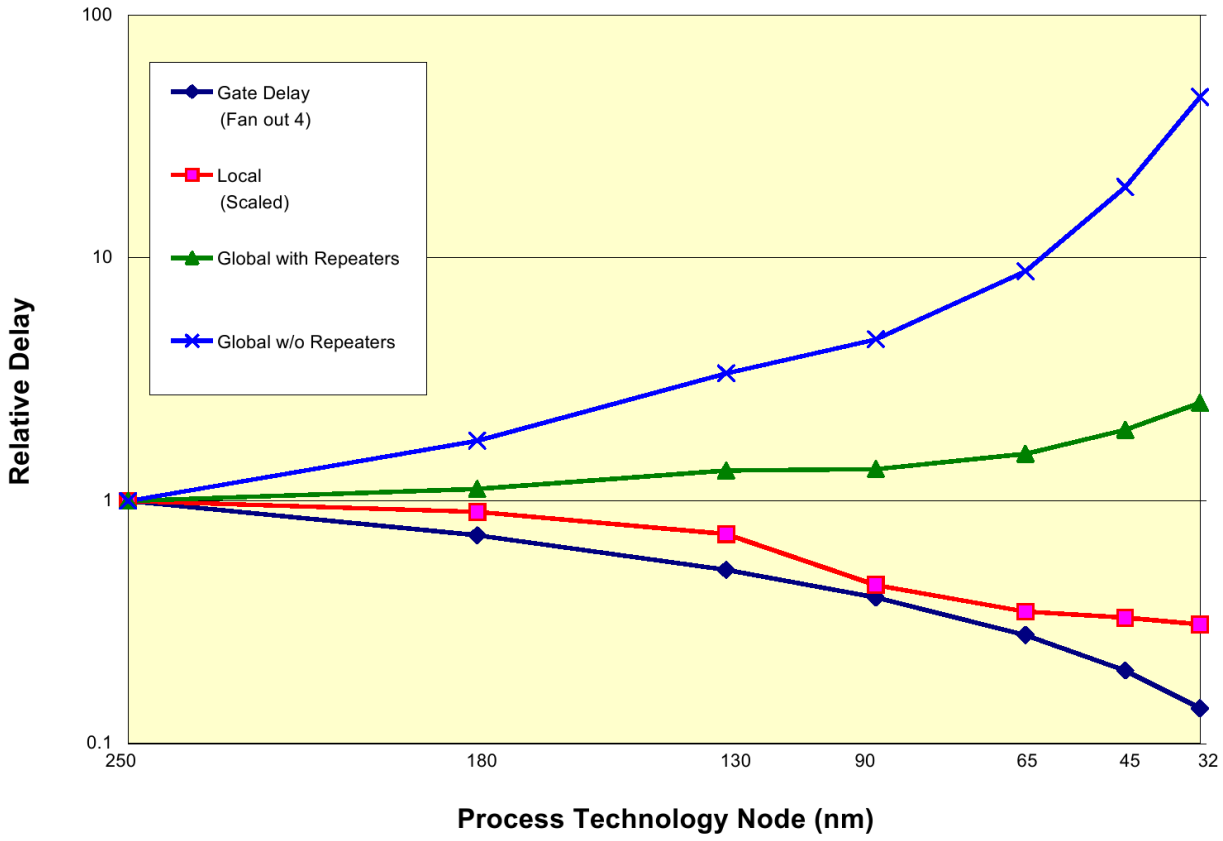


FIG. A.1 – Latences d'interconnection en fonction des processus de fabrication

ligne est de rajouter des répéteurs qui divisent le trajet en sections. Mais ceci a le contre-coup d'ajouter de la latence au système qui peut impacter sur son débit (en cas de cycles), et alors un nouveau problème est d'assurer la correction de la transformation, en rajoutant les répéteurs par rapport à la spécification synchrone initiale, ce qui peut conduire à un re-design des composants.

Le lecteur est invité à lire les manuscripts écrits par Christer Svensson [1] sur le sujet qui introduisent en détail la modélisation des fils, expliquent les problèmes et quelques solutions à propos des interconnections.

Les niveaux d'intégration disponibles aujourd'hui pour produire des systèmes sur puces sont tellement élevés qu'un système complet ne peut être implanté sur un seul ilôt synchrone. Les performances dépendent de manière critique de la latence des "longs" fils, comme l'ont montré un certain nombre d'études effectuées dans un passé proche [59, 74]. Malencontreusement, comme nous l'avons expliqué auparavant les délais sur les fils sont intrinsèquement inévitables et nous devons accepter cette nouvelle dimension dans la conception de systèmes sur puce complexes qu'est la latence.

Malgré tout, des systèmes de ce type montrent des problèmes qui sont connus des niveaux d'intégration de plus haut niveau. Comme nous l'avons introduit précedemment, il a été prédit qu'un signal aurait besoin de plus de cinq (et quelques fois plus de dix !) coups d'horloge pour traverser la puce entièrement (sans répéteur) comme l'illustre la figure A.1 [2] issue de la "roadmap" [7] effectuée par l'association ITRS.

Ainsi il est crucial de limiter les distances à traverser par les signaux critiques afin d'avoir des performances acceptables. Malheureusement, les données précises sur les longueurs des fils ne sont disponibles que très tard dans le flot de conception standard basé sur des modèles "statistiques" des fils et donc des re-designs coûteux et complexes sont nécessaires afin de satisfaire à la fois les contraintes de fonctionnalités et de performances.

Entre autre, la majorité des outils commerciaux utilisés pour fabriquer des puces utilisent un flot de conception synchrone qui justement ne peut pas prendre en compte directement cette information de latence. Néanmoins la majorité des fournisseurs d'outils ont pu s'adapter à ce problème en reliant algorithmes de synthèse logique avec la phase de placement/routage, et des modèles de fils beaucoup plus complexes qui prennent en compte la géométrie, le voisinage et plusieurs paramètres afin de fournir une évaluation plus précise du temps de propagation des fils (par exemple Physical Compiler® ou IC Compiler de Synopsys®) et égale-

[1] http ://www.ek.isy.liu.se/∼ *christer*/

[2] Figure issue du document : The International Technology Roadmap for Semiconductors, 2005 edition. SEMATECH :Austin, TX, 2005. Avec l'accord préalable du SEMATECH.

ment toutes les techniques ECOs (Engineering Change Orders) permettant d'effectuer des modifications incrémentales sur les designs [41]. Pour autant, malgré toutes ces améliorations le problème de latence demeure dans l'absolu.

Motivations

Ainsi lorsque nous développons des designs comportant plusieurs dizaines de millions de portes, nous souhaiterions disposer d'une méthode garantissant par construction que certaines propriétés sur le système soient satisfaites afin de disposer d'un système correct en peu de temps. En particulier, nous espérons pouvoir permettre au designer la possibilité de réutiliser des composants déja vérifiés et que cette composition n'altère aucunement les critères de correction.

Dans ce chapitre, nous présentons une théorie permettant de décrire des designs synchrones ne souffrant plus du syndrôme des *longs* fils. En utilisant cette méthodologie, le système peut être pensé comme complètement synchrone, c'est à dire un ensemble de modules communicants au travers de canaux ayant une latence d'un cycle d'horloge. Malheureusement, le *layout* final peut nécessiter plus d'un cycle d'horloge afin de retransmettre les signaux appropriés. Cette méthodologie ne nécessite pas des cycles coûteux de redesign ou de ralentir l'horloge. L'idée principale est issue du pipelining : nous partitionnons les longs fils en sections dont la longueur satisfait les contraintes de temps imposées par l'horloge en insérant des blocs logiques appelés *stations de relais,* qui ont une fonction similaire aux registres utilisés dans les pipelines. Les contraintes de temps sont imposées lors de la construction. Cependant, la latence d'un canal connectant deux modules peut coûter en cycles d'horloges. Si la fonctionnalité du design est basée sur l'ordre des signaux de sortie et non pas de leur temps exact, alors les modifications effectuées sur le design ne changeront pas la fonctionalité supposant que les composants du design sont *latency-insensitive* (insensibles à la latence), c'est à dire que le comportement de chaque module ne dépend pas de la latence induite par les canaux de communication.

Nous allons introduire ces concepts formellement et montrer les propriétés introduites précédemment.

Le chapitre suit le plan du papier théorique originel écrit par Carloni *et al.* dans un souci de couverture maximale, ce chapitre est organisé de la manière suivante : dans la prochaine section nous décrirons les fondations des designs latency-insensitive en introduisant la notion de processus *patient*. Dans la section d'après, nous discuterons comment un système de processus patients communicants au travers de canaux de communication pouvant être segmentés en introduisant des *relay stations* (stations de relais). Ensuite nous illustrerons l'ensemble de la méthodologie et discuterons sous quelles hypothèses un système générique peut être transformé en un processus patient. Après nous résumerons cette portion

théorique et nous étendrons vers les différentes extentions et travaux dérivants de cette théorie.

A.1.1 Insensibilité à la latence

Cette section utilise le modèle formel Tagged-Signal de Lee et Sangiovanni-Vincentelli [54] pour représenter les signaux et les processus.

Modèle Tagged-Signal

Maintenant nous allons introduire un modèle de Lee et Sangiovanni-Vincentelli [54], qui permet de décrire les traces d'exécutions effectuées par un système. Nous pouvons ainsi étudier tout type d'exécutions synchrones et asynchrones *conflict-free*. Ce modèle nous permettra ultérieurement de montrer des équivalences entre des exécutions synchrones et asynchrones dans le cadre des systèmes insensibles à la latence.

Étant donné un ensemble de *valeurs* V et un ensemble de *tags* que nous qualifierons d'"estampilles" T, un *évènement* E est un membre de $V \times T$. Deux évènements sont *synchrones* s'ils ont le même tag. Un *signal* est un ensemble d'évènements. Deux signaux sont synchrones si chaque évènement dans un signal est synchrone avec un évènement dans l'autre signal et réciproquement. Des signaux synchrones doivent avoir le même ensemble de tags.

L'ensemble de tous les n-uplets des signaux est noté S^N. Un processus P est un sous-ensemble de S^N. En particulier le n-uplet $s \in S$ satisfait le processus si $s \in P$. Un n-uplet s qui satisfait un processus est appelé un *comportement* du processus. Ainsi, un processus est un ensemble possible de comportements. Une *composition de processus* (aussi appelé *système*) $\{P_1,...,P_M\}$, est un processus définis comme une intersection de ces comportements notée $P = \bigcap_{m=1}^{M} P_m$. Puisque les processus peuvent être définis sur différents ensembles de signaux, pour former la composition nous avons besoin d'étendre l'ensemble des signaux sur lesquels chaque processus est défini afin de contenir tous les signaux de tous les autres processus. Notons que l'extension change le comportement des processus seulement formellement. Soit $J = (j_1,...,j_h)$ un ensemble ordonné d'entiers dans l'intervalle $[1,N]$, la projection d'un comportement $b = (s_1,...,s_N) \in S^N$ sur S^h est $proj_J(b) = (s_{j_1},...,s_{j_h})$. La projection d'un processus $P \subseteq S^N$ sur S^H est $proj_J(P) = (s' | \exists s \in P \wedge proj_J(s) = s')$. Une *connexion* C est un processus simple particulier où deux (ou plus) signaux dans le n-uplet sont contraints à être identiques : par exemple $C(i,j,k) \subset S^N : (s_1,...,s_N) \in C(i,j,k) \Leftrightarrow s_i = s_j = s_k$, avec $i,j,k \in [1,N]$. Dans un système *synchrone* chaque signal du système est synchrone

avec tous les autres signaux. Dans un système *timed*, l'ensemble des tags est un ensemble totalement ordonné. L'ordre sur les estampilles du signal s induit un ordre naturel sur l'ensemble des évènements de s.

Évènements informatifs et bloquants

Un système LI est un système synchrone *timed* dont l'ensemble des valeurs V est égal à $\Sigma \bigcup \{\tau\}$, où Σ est l'ensemble des symboles *informatifs* qui sont échangés entre les modules et $\tau \notin \Sigma$ est un symbole spécifique, représentant l"'absence" de signal informatif. A partir de maintenant, nous considérons tous les signaux comme supposés synchrones. L'ensemble des "estampilles" est supposé être en bijection avec l'ensemble des entiers naturels. Un évènement est appelé *informatif* s'il a un symbole informatif i_i comme valeur. Un évènement dont la valeur est un symbole τ est dit *bloquant*.

Définition 22. $\xi(s)$ note l'ensemble des évènements du signal s alors que $\xi_i(s)$ et $\xi_\tau(s)$ sont respectivements l'ensemble des évènements informatifs et bloquants. Le k-ième évènement (v_k, t_k) du signal s est noté $e_k(s)$. $\tau(s)$ est la notation de l'ensemble des estampilles du signal s, alors que $\tau_i(s)$ est l'ensemble des estampilles correspondantes aux évènements informatifs.

Les processus s'échangent des données *utiles* en envoyant et recevant des évènements informatifs. Idéalement seulement des évènements informatifs devraient être communiqués par les processus. Cependant, dans les systèmes LI, un processus peut ne pas avoir de donnée à cette estampille, nécessitant ainsi la sortie d'un évènement bloquant à la-dîte estampille.

Définition 23. L'ensemble de toutes les séquences d'éléments dans $\Sigma \bigcup \{\tau\}$ est noté Σ_{iat}. La longueur d'une séquence σ est $|\sigma|$ si elle est finie, autrement infinie. La séquence vide est noté ε et par définition $|\varepsilon| = 0$. Le i-ème terme de la séquence σ est noté σ_i.

Définition 24. La fonction $\sigma : S \times \tau^2 \to \Sigma_{iat}$ prend un signal $s = \{(v_0, t_0), ...\}$ et une paire ordonnée d'estampilles (t_i, t_j), $i \leq j$ et retourne une séquence $\sigma_{[t_i,t_j]} \in \Sigma_{iat}$ telle que $\sigma_{[t_i,t_j]}(s) = v_i, v_{i+1}, ..., v_j$. La séquence des valeurs du signal est notée $\sigma(s)$. La sous-séquence infinie de valeurs correspondantes à la séquence infinie des évènements partant de t_i est notée $\sigma_{[t_i,\infty]}(s)$.

Par exemple, considérons le signal $s = \{(i_1, t_1), (i_2, t_2), (\tau, t_3), (i_2, t_4), (i_1, t_5), (\tau, t_6)\}$ nous avons $\sigma(s) = i_1\ i_2\ \tau\ i_2\ i_1\ \tau$, $\sigma_{[t_2,t_4]}(s) = i_2\ \tau\ i_2$, $\sigma_{[t_5,t_5]}(s) = i_1$ et respectivement $|\sigma(s)| = 6$, $|\sigma_{t_2,t_4}(s)| = 3$, $|\sigma_{t_5,t_5}(s)| = 1$. Pour manipuler les séquences de valeurs nous définissons les opérations de filtrage suivantes.

Définition 25. $F_i : \Sigma_{iat} \rightarrow \Sigma^\star$ retourne une séquence $\sigma' = F_i[\sigma]$ telle que $\sigma'_k = \left\{\begin{array}{c}\sigma_{[t_k,t_k]}(s) \; si \; \sigma_{[t_k,t_k]}(s) \in \Sigma \\ \varepsilon \; autrement\end{array}\right\}$

Définition 26. $F_\tau : \Sigma_{iat} \rightarrow \{\tau\}^\star$ retourne une séquence $\sigma' = F_\tau[\sigma]$ telle que $\sigma'_k = \left\{\begin{array}{c}\sigma_{[t_k,t_k]}(s) \; si \; \sigma_{[t_k,t_k]}(s) = \tau \\ \varepsilon \; autrement\end{array}\right\}$

Par exemple, si $\sigma(s) = i_1\ i_2\ \tau\ i_2\ i_1\ \tau$, alors $F_i[\sigma(s)] = i_1\ i_2\ i_2\ i_1$ et $F_\tau[\sigma(s)] = \tau\ \tau$. Évidemment $|\sigma(s)| = |F_i[\sigma(s)]| + |F_\tau[\sigma(s)]|$. Les systèmes LI sont supposés avoir un horizon finis sur lequel des évènements informatifs apparaîssent, c'est à dire pour chaque signal s il y a une plus grande estampille $T \in T_i(s)$ qui correspond au *dernier* évènement informatif. Cependant pour construire cette théorie nous avons besoin d'étendre l'ensemble des signaux du système LI sur un horizon infini en ajoutant un ensemble d'estampilles tel que tous les évènements avec une estampille plus grande que T ont comme valeur τ.

Définition 27. Un signal s est strict si est seulement si tous les évènements informatifs précèdent tous les évènements bloquants, c'est à dire si et seulement s'il existe un $k \in N$ tel que $|F_\tau[\sigma_{[t_0,t_k]}(s)]| = 0$ et $|F_i[\sigma_{[t_k,t_\infty]}(s)]| = 0$. Un signal qui n'est pas strict est dit *retardé* ou *bloqué*.

Équivalence via Latence

Deux signaux sont équivalents via latence s'ils présentent la même séquence d'évènements informatifs, c'est à dire qu'ils sont identiques pour des retards différents entre deux évènements informatifs successifs.

Définition 28. Deux signaux s_1 et s_2 sont équivalents via latence noté $s_1 \equiv_\tau s_2$ si et seulement si $F_i[\sigma(s1)] = F_i[\sigma(s2)]$.

Le signal de référence s_{ref} d'une classe de signaux équivalents via latence est le signal strict obtenu par affectation de la séquence des valeurs informatives qui caractérise la classe d'équivalence de la première $|F_i[\sigma(s1)]|$ estampille. Par exemple, les signaux s_1 et s_2 présentent les séquences de valeurs suivantes :

$\sigma(s_1) = i_1\ i_2\ \tau\ i_1\ i_2\ i_3\ \tau\ i_1\ i_2\ \tau\ \tau\ \tau\ ...$
$\sigma(s_2) = i_1\ i_2\ \tau\ \tau\ i_1\ \tau\ i_2\ i_3\ \tau\ i_1\ \tau\ i_2\ \tau\ ...$

sont équivalents via latence. Leur signal de référence s_{ref} est caractérisé par la séquence de valeurs $\sigma(s_{ref}) = i_1\ i_2\ i_1\ i_2\ i_3\ i_1\ i_2\ \tau\ \tau\ \tau\ ...$

L'équivalence via latence contient la même séquence de valeurs informatives, mais avec des estampilles différentes. D'où, l'utilité d'identifier ces évènements

informatifs avec leur signal de référence : l'*ordinal* de ces évènements informatifs correspond avec la position dans le signal de référence.

Définition 29. L'ordinal d'un évènement informatif $e_k = (v_k, t_k) \in \xi_i(s)$ est défini comme $ord(e_k) = |F_i[\sigma[t_0, t_k](s)| - 1$. Soit s_1 et q_1 deux signaux équivalents via latence : deux évènements informatifs $e_k(s_1) \in \xi_i(s_1)$ et $e_l(q_1) \in \xi_i(q_1)$ sont dits des évènements correspondants si et seulement si $ord(e_k(s_1)) = ord(e_l(q_1))$. Le slack (la différence, le "mou") de deux évènements correspondants est définis comme $slack(e_k(s_1), e_l(q_1)) = |k - l|$.

Nous étendons la notion de comportements équivalents via latence sur des composants :

Définition 30. Deux comportements $(s_1, ..., s_N)$ et $(s'_1, ..., s'_N)$ sont équivalents si et seulement si $\forall i (s_i \equiv_\tau s'_i)$. Un comportement $b = (s_1, ..., s_N)$ est strict si pour tous les signaux $s_i \in b$ sont stricts. Toute classe de comportements équivalents via latence contient seulement un comportement strict qui est appelé comportement de référence.

Définition 31. Deux processus P_1 et P_2 sont équivalents via latence, noté $P_1 \equiv_\tau P_2$ si tout comportement de l'un est équivalent via latence d'un comportement de l'autre. Un processus P est strict si est seulement si pour tout comportement $b \in P$ est strict. Toute classe de processus équivalents via latence contient seulement un processus strict : le processus de référence.

Définition 32. Un signal s_1 est dominé via latence par s_2 noté $s_1 \leq_\tau s_2$ si et seulement si $s_1 \equiv_\tau s_2$ et $T_1 \leq T_2$ avec $T_k = max\{t | t \in T_i(s_k)\}, k = 1, 2$.

Donc, en se référant au précédent exemple, le signal s_1 est dominé par le signal s_2 puisque $T_1 = 9$ et $T_2 = 12$. Notons qu'un signal de référence est dominé via latence par tout signal appartenant à sa classe d'équivalence. La dominance via latence est étendue aux comportements et processus comme dans le cas de l'équivalence via latence. Un ordre total sur les évènements d'un comportement est nécessaire pour développer la théorie. En particulier, nous introduisons un ordre sur les évènements qui est motivé par la causalité : les évènements qui ont un ordinal plus petit sont ordonnancés avant ceux qui en ont un plus grand (penser à un processus strict où l'ordinal est relatif à l'estampille ; l'ordre implique que les évènements du passé ne dépendent pas des évènements futurs). De plus, pour éviter les comportements cycliques créés en traitant les évènements avec le même ordinal, nous supposons qu'il y existe un ordre sur les signaux. Cet ordre dans la spécification correspond aux dépendances entre les entrées et les sorties. Nous appliquons cette considération dans la forme la plus générale possible afin d'étendre son application à cette méthode.

Définition 33. Étant donné un comportement $b = (s_1, ..., s_N)$, $\leq_c$ note l'ordre bien formé sur l'ensemble des signaux. L'ordre bien formé induit un ordre lexicographique $\leq_{lo}$ sur l'ensemble des évènements informatifs de b, c'est à dire que pour toutes les paires d'évènements (e_1, e_2) avec $e_1 \in \xi_i(s_i)$ et $e_2 \in \xi_i(s_j)$.

$$e_1 \leq_{lo} e_2 \Leftrightarrow [(ord(e_1) < ord(e_2)) \vee ((ord(e_1) = ord(e_2)) \wedge (s_i \leq_c s_j))]$$

La fonction suivante retourne le premier évènement informatif (dans le signal s_j du comportement b) suivant un évènement $e \in b$ en respectant l'ordre lexicographique $\leq_{lo}$.

Définition 34. Étant donné un comportement $b = (s_1, ..., s_N)$ et un évènement informatif $e(s_i) \in \xi_i(s_i)$, la fonction *nextEvent* est définie par : $nextEvent = (s_j, e(s_i)) = min_{ek}(s_j) \in \xi_j(s_j)\{e(s_i) \leq_{lo} e_k(s_j)\}$.

Un *déplacement bloquant* reporte un évènement informatif d'un signal d'un comportement donné par une estampille. Le déplacement bloquant est utilisé pour prendre en compte les longs délais sur les fils c'est à dire que nous ajoutons des retards où c'est nécessaire pour garantir la correction fonctionnelle du design.

Définition 35. Étant donné un comportement $b = (s_1, ..., s_j, ..., s_N)$ et un évènement informatif $e_k = (v_k, t_k)$, un déplacement bloquant retourne un comportement $b' = stall(e_k(s_j)) = (s_1, ..., s'_j, ..s_N)$ tel que pour tout $l \in [1, N]$ $\sigma_{[t_0,t_{k-1}]}(s'_j) = \sigma_{[t_0,t_{k-1}]}(s_j), \sigma_{[t_k,t_k]}(s'_j) = \tau$, $\sigma_{[t_{k+l+1},t_{k+l+1}]}(s'_j) = \sigma_{[t_{k+l},t_{k+l}]}(s_j)$.

L'effet du déplacement bloquant est un effet *"retardateur"* (les auteurs dans le papier originel utilise le terme *procrastination* qui désigne une tendance chronique à retarder) sur les autres signaux du comportement b en correspondance avec l'évènement suivant $e_k(s_j)$ dans l'ordre lexicographique. Le processus "répondra" à cette insertion d'un blocage en retardant les autres signaux en relation de causalité avec ce premier.

Définition 36. Un effet *" retardateur"* est une injection qui prend un comportement $b' = (s'_1, ..., s'_N) = stall(e_k(s_j))$ résultant de l'application d'un déplacement bloquant sur l'évènement $e_k(s_j)$ du comportement $b = (s_1, ..., s_N)$ et retourne un ensemble de comportements $P\xi[stall(e_k(s_j))]$ tel que $b'' = (s''_1, ..., s''_N) \in P\xi[b']$ si et seulement si :

- $s''_j = s'_j$.
- $\forall i \in [1, N], i \neq j, s''_i \equiv_\tau s'_i$ *and* $\sigma_{[t_0,t_{l-1}]}(s''_i) = \sigma_{[t_0,t_{l-1}]}(s'_i)$ où t_l est l'estampille de l'évènement $e_l(s_i) = nextEvent(s_i, e_k(s_j))$.
- $\exists K$ tel que $\forall i \in [1, N], i \neq j, \exists k_i \leq K, \sigma_{[t_{l+k_i},\infty]}(s''_i) = \sigma_{[t_l,\infty]}(s'_i)$.

Chaque comportement dans $P\xi[b']$ est obtenu de b' en insérant lorsque c'est possible un autre évènement de bloquage dans n'importe quel signal de b', mais seulement aux "dernières" estampilles, c'est à dire que nous retardons les évènements informatifs qui suivent $e_k(s_j)$ en respectant l'ordre lexicographique $\leq_{lo}$. Nous observons que l'effet retardateur retourne un comportement qui domine en latence le comportement originel.

Processus patients

Nous pouvons maintenant définir la notion de processus patient : un processus patient peut effectuer des déplacements bloquants sur n'importe quel signal de ces comportements en réagissant avec les effets retardateurs appropriés. La patience est la condition clef pour combiner les blocs d'IP en utilisant cette méthode. Les théorèmes suivants garantissent que pour un processus patient, la notion d'équivalence de latence de processus est compositionnelle.

Définition 37. Un processus est patient si et seulement si : $\forall b = (s_1, ..., s_N) \in P, \forall j \in [1, N], \forall e_k(s_j) \in \xi_i(s_j), (P\xi[stall(e_k(s_j))] \bigcap P \neq \emptyset)$.

Donc, le résultat d'un déplacement bloquant sur un des évènements du processus patient peut ne pas satisfaire le processus, mais un de ces comportements utilisant l'effet retardateur satisfait le processus, c'est à dire que si nous bloquons un signal sur une entrée du bloc fonctionnel, le bloc sera forcé de retarder certaines de ses sorties ou si nous demandons un signal de sortie d'être retardé alors un retard approprié devra être ajouté aux entrées.

Lemme 19. *Soit P_1 et P_2 deux processus patients. Soit $b_1 \in P_1, b_2 \in P_2$ deux comportements avec le même ordre lexicographique c'est à dire $b_1 \equiv_\tau b_2$. Alors, il existe un comportement $b' \in (P_1 \bigcap P_2)$, $b_1 \equiv_\tau b' \equiv_\tau b_2$.*

Théorème 20. *Si deux processus P_1 et P_2 sont des processus patients alors $P_1 \bigcap P_2$ est un processus patient.*

Théorème 21. *Pour tout processus patient P_1, P_2, P_1', P_2' si $P_1 \equiv_\tau P_1'$ et $P_2 \equiv_\tau P_2'$ alors $(P_1 \bigcap P_2) \equiv_\tau (P_1' \bigcap P_2')$.*

Ainsi, nous pouvons remplacer n'importe quel processus dans un système avec processus patients par un processus équivalent via latence. Un théorème similaire existe pour remplacer des processus stricts par des processus patients.

Théorème 22. *Pour tout processus strict P_1, P_2 et tout processus patient $P_1', ' P_2'$, si $P_1 \equiv_\tau P_1'$ et $P_2 \equiv_\tau P_2'$ alors $(P_1 \bigcap P_2) \equiv_\tau (P_1' \bigcap P_2')$.*

Cela signifie que nous pouvons remplacer tous les processus d'un système de processus stricts par des processus patients correspondants et le système résultant sera alors équivalent via latence. C'est la notion clef de l'idée du LI : prendre un design basé sur l'hypothèse que nous avons des blocs fonctionnels et que les communications "ne prennent pas de temps" (hypothèse synchrone), c'est à dire que les processus correspondant aux blocs fonctionnels et leurs compositions sont stricts, nous les remplaçons avec un design où les communications prennent du temps (plus d'un cycle d'horloge, un multiple) et du coup les signaux sont retardés mais sans altération de la séquence des évènements informatifs observés au niveau du système, c'est à dire un ensemble de processus patients.

A.1.2 LID

Comme expliqué dans l'introduction du chapitre, le but de la méthodologie LID est d'être capable de "pipeliner" des canaux de communication en insérant un nombre arbitraire d'éléments de stockage, de mémorisation. Dans le cadre de cette théorie, cette opération correspond à ajouter un processus particulier appelé *relay station* (station de relais) au système donné. Dans cette section, nous allons premièrement montrer comment les systèmes patients (composés de processus patients) sont insensibles à l'insertion de stations de relais et, ensuite nous discuterons sous quelle hypothèse un système en général peut être transformé en système patient.

Canaux et Tampons

Un *canal* est une connexion (cf. section précédente) contraignant deux signaux à être identiques.

Définition 38. Un canal $C(i,j) \subset S^N, i,j \in [1,N]$ est un processus c'est à dire $b = (s_1,...,s_N) \in C(i,j) \Leftrightarrow s_i = s_j$.

Lemme 23. *Un canal $C(i,j) \subset S_N$ n'est pas un processus patient.*

Définition 39. Un tampon $B^c_{l_f,l_b}(i,j)$ avec une capacité $c \geq 0$, de latence minimale avant $l_f \geq 0$ et latence minimale arrière $l_b \geq 0$ est un processus tel que $\forall i,j \in [1,N] : b = (s_1,..,s_N) \in B^c_{l_f,l_b}(i,j)$ si et seulement si $(s_i \equiv_\tau s_j)$ *et* $\forall k \in \mathbb{N}$

$$0 \leq |F_i[\sigma_{[t_0,t_{(k-l_f)}]}(s_i)]| - |F_i[\sigma_{[t_0,t_k]}(s_j)]| \text{ (1)}$$
$$c \geq |F_i[\sigma_{[t_0,t_k]}(s_i)]| - |F_i[\sigma_{[t_0,t_{(k-l_b)}]}(s_j)]| \text{ (2)}$$

Par définition étant donné une paire d'indexes $i,j \in [1,N]$ pour tout $l_b, l_f, c \geq 0$, tous les tampons $B^c_{l_f,l_b}(i,j)$ sont équivalents via latence. Observons également que

le tampon $B^0_{0,0}(i,j)$ coïncide avec le canal $C(i,j)$. En particulier, nous sommes intéressés par les tampons ayant une latence unitaire et nous voulons établir sous quelles conditions de tels tampons sont des processus patients.

Théorème 24. *Soit $l_b = l_f = 1$, pour tout $c \geq 1, B^c_{1,1}$ est patient si et seulement si $s_i \leq_c s_j$.*

Considérons un système strict $P_{strict} = \bigcap_{m=1}^{M}$ avec N signaux stricts $s_1, ..., s_N$. Comme expliqué dans la section précédente les processus peuvent être définis sur différents ensembles de signaux et pour les composer nous avons besoin d'étendre formellement l'ensemble des signaux pour chaque processus pour contenir tous les signaux de tous les processus. Cependant et sans perte de généralité, considérons le cas particulier de composer M processus qui sont déjà définis sur les mêmes signaux N. Donc, n'importe quel comportement générique $b_m = (s_{m_1}, ..., s_{m_N})$ de P_m est aussi un comportement de P_{strict} si et seulement si $\forall l \in [1,M], l \neq m$ processus P_l contient un comportement $b_l = (s_{l_1}, ..., s_{l_N})$ tel que $\forall n \in [1,N](s_{l_n} = s_{m_n})$. En fait, nous pouvons faire l'hypothèse de dériver le système P_{strict} en connectant les M processus avec $(M-1).N$ processus canaux $C(l_n, (l+1)_n)$, où $l \in [1,(M-1)]$ et $n \in [1,N]$. De plus, nous pouvons aussi "décomposer" chaque processus canal $C(m_n, l_n)$ avec un nombre arbitraire X de processus canaux $C(m_n, x_1), C(x_1, x_2), ..., C(x_{X-1}, l_n)$, en ajoutant $X-1$ signaux auxiliaires, chacun d'entre eux forcés d'être égal à $m_n = l_n$. La théorie développée dans la section précédente garantie que si nous remplaçons chaque processus $P_m \in P_{strict}$ avec un processus patient équivalent via latence et chaque canal $C(i,j)$ avec un buffer patient $B^1_{1,1}(i,j)$ nous obtenons un système $P_{patient}$ qui est un système patient équivalent via latence à P_{strict}. En fait, avoir un tampon patient dans un système patient est équivalent à avoir un canal dans un système strict. Puisque la "décomposition" d'un canal $C(i,j)$ n'a pas d'effet observable sur un système strict, nous pouvons alors librement ajouter n'importe quel nombre arbitraire de tampons patients dans le système patient correspondant pour remplacer le canal. Puisque nous utilisons des tampons patients avec des latences unitaires, nous pouvons les distribuer sur les longs fils sur le système sur puce qui implémente $C(i,j)$ d'une telle manière que le fil soit décomposé en segments dont les longueurs physiques peuvent être traversées en un seul cycle d'horloge physique.

Stations de Relais

Le lemme 19 montre qu'aucun comportement dans $B^1_{1,1}(i,j)$ ne peut contenir deux évènements informatifs de s_i, s_j qui sont synchrones : cela implique que le débit maximum obtenable est de $0,5$ qui est loin d'être optimal. À la place, un

tampon $B^2_{1,1}(i,j)$ est le tampon de capacité minimum qui est capable de "transférer" une unité d'information par estampille, autorisant alors dans le meilleur des cas de communiquer avec un débit maximal de 1. Illustration de deux comportements possibles de ce type de tampon :

$$B^1_{1,1} = \left\{ \begin{array}{l} s_1 = i_1\ \tau\ i_2\ \tau\ i_3\ \tau\ i_4\ \tau\ \tau\ \tau\ i_5\ \tau\ i_6\ \tau\ i_7\ \tau\ i_8\ \tau\ i_9\ \tau\ \tau\ \tau\ i_{10}\ \tau \\ s_2 = \tau\ i_1\ \tau\ i_2\ \tau\ i_3\ \tau\ i_4\ \tau\ \tau\ \tau\ i_5\ \tau\ i_6\ \tau\ i_7\ \tau\ i_8\ \tau\ \tau\ \tau\ i_9\ \tau\ i_{10}\ \tau \end{array} \right\}$$

$$B^2_{1,1} = \left\{ \begin{array}{l} s_1 = i_1\ i_2\ i_3\ \tau\ \tau\ i_4\ i_5\ i_6\ \tau\ \tau\ \tau\ i_7\ \tau\ i_8\ i_9\ i_{10} \\ s_2 = \tau\ i_1\ i_2\ i_3\ \tau\ \tau\ i_4\ \tau\ \tau\ \tau\ i_5\ i_6\ i_7\ \tau\ i_8\ i_9\ i_{10} \end{array} \right\}$$

Lemme 25. *$B^2_{1,1}(i,j)$ est le tampon de capacité minimale avec $l_f = l_b = 1$ tel que*

$$\exists b^\star = (s^\star_1, ..., s^\star_N) \in B^2_{1,1}(i,j) \wedge \exists k \in \mathbb{N}, (e_k(s^\star_i) \in \xi_i(s^\star_i) \wedge (e_k(s^\star_j) \in \xi_i(s^\star_j))$$

Définition 40. Le tampon $B^2_{1,1}$ est appelé une station de relais (RS).

A.1.3 Méthodologie LID

Dans cette section nous allons aller vers l'implantation de la théorie introduite dans les sections précédentes. Nous faisons les hypothèses suivantes :

- les blocs fonctionnels sont des processus synchrones.
- il y a un ensemble de signaux pour chaque processus qui sont considérés comme des entrées au processus et réciproquement un ensemble de signaux de sorties. Les processus sont des fonctions.
- les processus sont strictements causaux (un processus est strictement causal si deux sorties peuvent être différentes à des estampilles qui suivent strictement les estampilles des entrées produites).
- les processus appartenant à une classe particulière de processus dénommés "bloquables", une condition nécessaire que chaque processus doit implanter.

Les idées de base sont les suivantes : composer un ensemble d'IPs synchrones de la manière la plus efficace est aisée si nous supposons que les hypothèses synchrones s'appliquent. Cette composition correspond à une composition de processus stricts puisqu'il n'y a pas besoin d'insérer à priori d'évènements bloquants. Cependant, nous avons argumenté dans l'introduction que cette hypothèse synchrone n'est pas vraisemblable au niveau des communications. Si les processus à composer sont patients, alors ajouter un nombre approprié de stations de relais donnera un processus équivalent via latence à la composition stricte. Donc, si nous utilisons la définition qu'un comportement correct est le fait que la séquence des évènements informatifs ne change pas, l'ajout de stations de relais résoud le problème. Cependant, cela nécessite que les processus soient patients ce qui est une

hypothèse forte pour débuter. Cependant, en pratique un système patient peut être dérivé d'un processus strict de la manière suivante : premièrement, nous prenons chaque processus strict P_m et nous le composons avec un ensemble de processus auxiliaires afin d'obtenir un processus équivalent patient P'_m. Pour être capable de faire cela, tous les processus P_m doivent satisfaire une condition simple (les processus doivent être bloquables) spécifiée dans la prochaine section. Après, nous mettons tous les processus patients ensemble en les connectant entre eux via des stations de relais. L'ensemble des processus auxiliaires implémentent un mécanisme semblable à une "queue" entre les signaux de P_m d'une telle manière que les évènements informatifs sont mis dans des tampons et réordonnés avant d'être passés à P_m : les évènements informatifs ayant le même cardinal sont passés à P_m synchronement. Dans le manuscrit, nous introduisons d'abord une définition formelle des processus fonctionnels. Après nous présentons une notion simple de processus bloquable et nous prouvons que le processus bloquable peut être encapsulé dans un processus "wrapper" qui offre une interface au protocole insensible à la latence.

Processus bloquables

Une *entrée* à un processus $P \subseteq S^N$ est une contrainte externe imposée $P_I \subseteq S^N$ telle que $P_I \bigcap P$ est un ensemble total de comportements acceptables. En général nous considérons que les processus ont des signaux d'entrées et de sorties : dans ce cas, étant donné un processus P, l'ensemble des signaux peut être partitionné dans trois sous-ensembles disjoints en partitionnant l'ensemble des indexes $\{1,...,N\} = I \bigcup O \bigcup R$, où I est l'ensemble ordonné des indices des signaux d'entrées de P, O est l'ensemble ordonné des signaux de sorties de P et R est l'ensemble ordonné des autres signaux restants (aussi appelés signaux locaux). Un processus est *fonctionnel* par rapport à (I,O) si pour tous les comportements $b \in P$ et $b' \in P$, $proj_I(b) = proj_I(b')$ implique $proj_O(b) = proj_O(b')$. Dans le manuscrit, nous considérons seulement les processus strictement causaux et pour chacun d'eux nous supposons que l'ordre bien fondé $\leq_c$ de la définition 33 subsume les relations de causalité des signaux, c'est à dire formellement.

Définition 41. Un processus P avec $I = \{1,...,Q\}$ et $O = \{Q+1,...,N\}$ est bloquable si et seulement si pour tout $b = (s_1,...,s_Q,s_{Q+1},...,s_N) \in P$ et pour tous les $k \in \mathbb{N}$:

$$\forall i \in I(\sigma_{[t_k,t_k]}(s_i) = \tau) \Leftrightarrow \forall j \in O(\sigma_{[t_{k+1},t_{k+1}]}(s_j) = \tau)$$

Donc, alors qu'un processus patient tolère des distributions arbitraires d'évènements bloquants dans ses signaux (tant que la causalité est préservée), un processus bloquable demande des "patterns" plus réguliers : les symboles τ peuvent

être insérés synchronement (c'est à dire sur la même estampille) sur tous les signaux d'entrée et cette insertion implique l'insertion de symboles τ sur toutes les sorties à l'estampille considérée. Supposer qu'un processus fonctionel est bloquable est une hypothèse assez raisonnable vis à vis des possibilités d'implantations pratiques. En fait, la plupart des systèmes matériels peuvent être bloqués : par exemple, considérons n'importe quel bloc de logique séquentielle qui a une *gated clock* ou une machine à état finis de Moore M avec une entrée supplémentaire, qui, si égale à τ force M à rester dans l'état courant et d'émettre τ au prochain cycle.

Encapsulation de processus bloquants

Maintenant, notre but est de définir un groupe de processus fonctionnels qui peuvent être composés avec des processus bloquables P pour dériver un processus patient qui est équivalent modulo latence à P . Nous commençons par considérer un processus qui aligne tous les évènements informatifs d'un ensemble de canaux.

Définition 42. Un "égaliseur" est un processus E avec $I = \{1,..,Q\}$ et $O = \{Q+1,...,2.Q\}$, c'est à dire que tous les comportements $b = (s_1,...,s_Q,s_{Q+1},...,s_{2.Q}) \in E : \forall i \in I\ ,\ (s_i \equiv_\tau s_{Q+i})$ et $\forall k \in \mathbb{N}$

$$\forall i,j \in O((\sigma_{[t_k,t_k]}(s_i) = \tau) \Rightarrow (\sigma_{[t_k,t_k]}(s_j) = \tau))$$
$$min_{i\in I}\{|F_i[\sigma_{[t_0,t_k]}(s_i)]|\} - max_{j\in O}\{|F_i[\sigma_{[t_0,t_k]}(s_j)]|\} \geq 0$$

La première relation force les signaux de sortie à avoir des évènements bloquants seulement synchronement, alors que la seconde garantit qu'à toute estampille le nombre des évènements informatifs apparaissant à n'importe quelle entrée est toujours plus grand que le nombre des évènements informatifs apparaissant sur n'importe quelle sortie (respect de la causalité, l'ordre). En particulier, la présence d'évènements bloquants à n'importe quelle sortie à une estampille donnée force la présence d'un évènement bloquant sur toutes les sorties à la même estampille. L'exemple suivant illustre un comportement possible d'un "égaliseur".

Exemple de comportement d'un égaliseur E avec un comportement $I = \{1,2,3\}$ et $O = \{4,5,6\}$.

$$s_1 = i_1\ i_3\ i_1\ \tau\ i_3\ \tau\ \tau\ ...\ \Rightarrow s_4 = \tau\ i_1\ \tau\ i_3\ i_1\ \tau\ i_3\ ...$$
$$s_2 = \tau\ i_4\ \tau\ i_7\ i_8\ \tau\ i_8\ ...\ \Rightarrow s_5 = \tau\ i_4\ \tau\ i_7\ i_8\ \tau\ i_8\ ...$$
$$s_3 = \tau\ i_5\ i_5\ \tau\ i_9\ i_6\ ...\ \Rightarrow s_6 = \tau\ i_5\ \tau\ i_5\ i_9\ \tau\ i_6\ ...$$

Définition 43. Une station de relais étendue ξRS est un processus avec $I = \{i\}$ et $O = \{j,l\}$, $i \neq j \neq l$ tel que les signaux s_q, s_2 sont apparentés par les inégalités (1) et (2) de la définition 39 avec ($l_f = l_b = 1$ et $c = 2$) et $\forall k \in \mathbb{N}$:

$$\sigma_{[t_k,t_k]}(s_l) = \left\{ \begin{matrix} 1 \; si \; |F_i[\sigma_{[t_0,t_k]}(s_i)]| - |F_i[\sigma_{[t_0,t_{k-1}]}]| = 2 \\ 0 \; autrement \end{matrix} \right\}$$

Définition 44. Un générateur de signal bloquant *SSG* est un processus avec $I = \{1,...,Q\}$ et $O = \{Q+1\}$ tel que $\forall b = (s_1,...,s_{Q+1}), \forall k \in \mathbb{N}, \forall i \in [1,Q], (F_i[\sigma_{t_k,t_k}](s_i) \in [0,1])$ et

$$\sigma_{[t_k,t_k](s_{Q+1}} = \left\{ \begin{matrix} \tau \; si \; \exists j \in [1,Q](F_i[\sigma_{t_k,t_k}](s_j) = 1) \\ 0 \; autrement \end{matrix} \right\}$$

Comme illustré dans la figure A.2, n'importe quel processus P peut être décomposé avec un "égaliseur", un générateur de signaux bloquants et quelques stations de relais étendues afin de dériver un processus patient équivalent via latence à P.

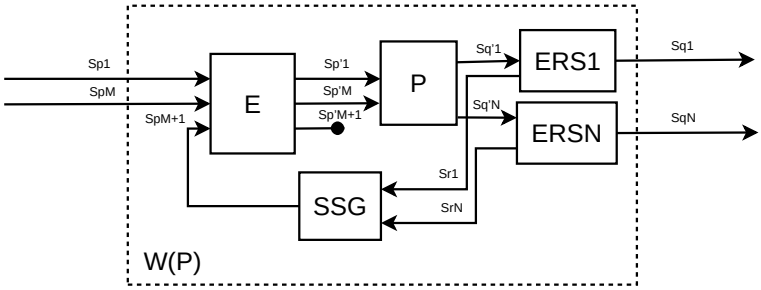


FIG. A.2 – Encapsulation d'un processus bloquable dans un "wrapper"

Définition 45. Soit P un processus bloquable avec $I_P = \{p'_1,...,p'_M\}$ et $O_P = \{q'_1,...,q'_N\}$. Un processus "wrapper" (ou "shell") $W(P)$ de P est le processus avec $I_W = \{p_1,...,p,m\}$ et $O_W = \{q_1,...,q_N\}$ qui est obtenu en composant P avec les processus suivants :

- un "égaliseur" E avec $I_E = \{p_1,...,p_M,p_{M+1}\}$ et $O_E = \{p'_1,...,p'_M,p'_{M+1}\}$,
- N stations de relais étendues $\xi RS_1, \xi RS_2,..., \xi RS_N$ telles que $I_j = \{q'_j\}$ et $O_j = \{q_j, r_j\}$ avec $j \in [1,N]$
- un générateur de signal bloquant *SSG* with $I_G = \{r_1,...,r_N\}$ et $O_G = \{p_{M+1}\}$.

Théorème 26. *Soit $W(P)$ le processus "wrapper" de la définition précèdente. Le processus $W = proj_{I_W \cup O_W}(W(P))$ est un processus patient qui est équivalent via latence à P.*

En conclusion, la méthodologie LID peut être résumée de la manière suivante :

1. Commencer avec un système de M processus bloquables et N canaux.

2. Encapsuler chaque processus bloquable pour obtenir un processus "wrapper".
3. Utiliser des stations de relais pour décomposer chaque canal en segments dont les longueurs physiques peuvent être traversées en un seul cycle d'horloge.

Cette approche "orthogonalise" clairement calculs et communications : en fait, nous pouvons construire un système en mettant ensemble des "cores" (qui peuvent être arbitrairement aussi complexes que nous le souhaitons à condition de respecter l'hypothèse d'être bloquable) et des "wrappers" (qui s'interfacent avec les canaux, en "parlant" le dialecte du protocole insensible à la latence). Bien que la fonctionnalité spécifique du système est distribuée dans les "cores", les wrappers peuvent être générés automatiquement autour d'eux [3]. Finalement, la validation du système peut maintenant être effectivement décomposée et basée sur un raisonnement de type "assume-guarantee" : chaque wrapper est vérifié en supposant un protocole donné, et le protocole est vérifié séparement.

Conclusions Préléminaires

Une nouvelle méthodologie de design pour de grands systèmes sur puce a été introduite. Cette méthodologie est basée sur l'hypothèse que le design est construit par assemblage de blocs d'IP qui ont été construits et vérifiés auparavant. Le but principal est de développer une théorie pour la composition des blocs d'IP qui *assurent* la correction du design complet. Le point principal porte sur les propriétés temporelles puisque les nouvelles générations de processus de fabrication des puces souffrent (et continueront de souffrir encore plus) des délais sur les longs fils qui causent toujours des redesigns coûteux, malgré le nombre impressionant de progrès qui ont été faits durant cette dernière décennie. Les designs créés par cette méthodologie sont appelés des designs Latency Insensitive Design (LID). LID sont des systèmes synchrones distribués et qui sont réalisés par assemblage de modules fonctionnels échangeant des données sur des canaux de communication en suivant un protocole de communication insensible à la latence. Le protocole garantit que les designs LID composés de modules fonctionnellement corrects, se comportent indépendemment des délais des fils. Cela nous permet de pipeliner les longs fils en insérant des éléments spéciaux de mémorisation appelés stations de relais. Ce protocole fonctionne sur l'hypothèse que les blocs fonctionnels satisfont certaines propriétés nécessaires.

[3] C'est pour cette raison que les wrappers s'appellent des *coquilles* (shells) car ils "protègent" la perle (pearl) des "soucis" de l'architecture externe de communication.

A.1.4 Extensions

Cette section introduit l'ensemble des extensions (généralement du point de vue d'implantation) effectuées sur les systèmes LID par un certain nombre de groupe de recherches. Nous présentons d'abord la méthode *"recycling"* de Carloni pour limiter l'utilisation du protocole de "back-pressure". Ensuite nous verrons la technique trouvée par Casu et Macchiarulo utilisant des algorithmes issus du SoftwarePipelining permettant de supprimer le protocole de synchronisation en employant un ordonnancement périodique et remplaçant les stations de relais par des simples registres. Puis Synchronous Latency Insensitive mise au point par Svensson qui utilise des FIFOs "élastiques" afin de se passer du protocole de synchronisation et effectuer en même temps la synchronisation. Puis nous discuterons des travaux effectués par Singh et Theobald afin de généraliser le LID vers des architectures multi-horloges. Après nous présenterons une implantation détaillée de LID effectuée aussi par Casu et Macchiarulo. Enfin nous discuterons du rôle de la back-pressure sur la base d'un papier de Carloni [20]. Puis nous introduierons les travaux effectués par Suhaib *et. al* [72, 71] afin de vérifier formellement le protocole du LID.

Approche "recycling" de Carloni

Un des problèmes majeur de l'approche LID est de ne pas garantir la performance du design, à cause des signaux de bloquage qui peuvent arriver sporadiquement et dont les variations arbitraires des latences sur les interconnections sont une des causes. Le papier [23] montre une méthode simple permettant d'analyser l'impact des latences des interconnections sur le débit global du système, et introduit une technique altérant la topologie du système afin de limiter le recours au protocole de synchronisation et donc à la génération de bloquages. Dans le cas d'un DAG, il n'y a pas de problème : le débit est de 1 ; par contre lorsque nous considèrons un système disposant d'au moins une partie fortement connexe, son débit est alors borné par le cycle le lent, comme l'a montré Reiter dans les années soixante [68]. Le problème de déterminer quel est le débit maximum obtensible sur un tel système est dénommé "maximal cycle mean" dans la littérature. Il est résolu par un certain nombre d'algorithmes plus ou moins efficaces en pratique comme l'illustre le papier d'Ali Dasdan [36]. Le but de Carloni est de modifier la topologie en rajoutant des stations de relais sur des cycles non critiques afin d'amener ces cycles au rythme du plus lent. Cela permet de limiter le recours au protocole de synchronisation, par contre il peut y avoir des "résidus" car la solution est rationnelle. Pour combler ce problème, nous utilisons alors dans ce cadre le protocole de synchronisation. Dans un autre papier [22], Carloni *et al.* utilisent à la fois la technique précédente en conjonction avec la technique de *retiming*,

une optimisation classique s'appliquant sur la logique permettant de minimiser la période de l'horloge d'un circuit synchrone [55] (le retiming est une technique d'optimisation clef dans tout outil de synthèse logique qu'il soit académique ou commercial). La limitation inhérente dans cette technique est due à un invariant, ainsi certains algorithmes appartenant au software pipelining sont capables d'être plus performants car ils n'ont pas cette contrainte. Cet invariant se décrit de la manière suivante : le nombre de registres dans n'importe quel cycle C est constant lors de la transformation.) Il faut se souvenir que la technique "recycling" a un coup qui peut être important en surface et latence, entre autre il y a des cas où une telle optimisation n'est pas convenable au niveau des performances temporelles (par rapport à d'autres techniques comme l'ordonnancement statique). Ainsi même avec un couplage avec le retiming nous n'obtiendrons pas forcément de bonnes performances.

Approche de Casu et Macchiarulo

L'approche de Casu et Macchiarulo [28] montre qu'il est possible de remplacer le protocole de synchronisation utilisé dans Latency Insensitive en employant un algorithme d'ordonnancement s'appliquant sur l'horloge des blocs fonctionnels, permettant ainsi de réduire grandement les ressources utilisées par le routage, autorisant ainsi une réduction substancielle de la surface utilisée pour pipeliner les communications. Le protocole est coûteux à implanter car il est nécessaire de rajouter deux signaux *valid* et *stop* à tous les canaux de communication. De tels signaux augmentent les besoins en fils causant ainsi un renforcement du problème de congestion déjà existant au niveau de ces fils. Entre autre, l'insertion des stations de relais le long de ces interconnections pose de sérieuses contraintes puisque cette dernière impose de placer au moins deux registres et une petite machine à états finis pour chacune d'entre elles. Les auteurs ont montré par l'utilisation d'un algorithme adapté qu'ils peuvent trouver un ordonnancement optimal au niveau du débit, qui borne le système à son cycle le plus lent. Entre autre leur technique permet de remplacer les coûteuses stations de relais par de simples répéteurs. Les signaux du protocole sont alors inutiles, ainsi l'économie sur les ressources de routage est importante.

Dans l'approche synchrone il n'y a pas de latence, il en résulte alors qu'à tous les cycles d'horloges nous lisons toutes les entrées et nous produisons toutes les sorties : le système n'a pas de problème de performance et son débit est de 1. Dans le cas de LID, en général nous ne pouvons atteindre ce débit optimal à cause de l'introduction de la métrique latence. Pour préserver la "synchronicité" nous pouvons contrôler les horloges des éléments de façon à ce que leur horloge ne soit active que lorsque les données valides sont arrivées. Cela nécessite un ordonnancement global des horloges afin que les différentes unitées soient coordonnées.

En analysant la topologie nous pouvons en déduire la borne maximale atteignable par le système pour ses performances en débit, il n'est pas utile d'effectuer de simulation. L'implantation de l'ordonnanceur est effectuée grâce à des registres à décalage comme l'illustre la figure A.3 avec une séquence de 1 et 0 dénotant respectivement les instants où l'horloge est active et inactive. La sortie de ce registre est utilisée pour générer l'horloge du bloc fonctionnel de manière similaire à ce qu'effectue le shell. La sortie du shell, est envoyée à un autre shell ou à un registre. L'exemple de la figure A.3 montre deux shells communicants au travers d'un registre et qui sont actifs deux fois sur trois, nous nous apercevons que l'ordonnancement du deuxième shell est celui du premier décalé de deux instants. (Cette figure est conceptuelle, un clock-gating n'est pas effectué en pratique avec une porte ET). La difficulté maintenant est de calculer cet ordonnancement de manière effective. L'ordonnancement doit être valide, c'est à dire qu'il doit satisfaire deux propriétés : ne pas perdre ou écraser de données, et l'absence de duplication de données. En d'autres termes l'ordonnancement doit garantir les propriétés de sûreté énoncées dans la théorie du LID. Nous ajoutons aussi la contrainte de performance énoncée plus haut à propos du débit. Pour ce faire, Casu et Macchiarulo utilisent une variante de l'algorithme mis au point par Boyer *et al.* dans [67] lui même dérivé d'un algorithme de la famille du software pipelining créé par Van Dongen *et al.* [37]. Ensuite ils introduisent très sommairement le problème de *l'égalisation* (ainsi que dans le papier [29]) dans le cas des parties fortement connexes où des branches de longueur différentes reconvergent, où une branche plus courte en latence oblige à insérer des bulles (absences de donnée) afin de ne pas arriver trop tôt. Mais dans le cas général cela n'est pas possible car malheureusement la solution du système d'un tel système est fractionnaire et l'insertion de registre correspond à un entier. Ils montrent un exemple de circuit permettant d'effectuer ce genre de synchronisation, confère la figure A.4. Ainsi qu'un autre circuit qui permet d'effectuer une initialisation du système en utilisant un registre à décalage en amont de l'ordonnanceur ; voir la figure A.5.

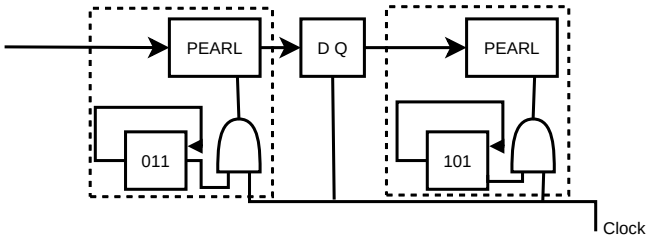


FIG. A.3 – Implantation ordonnanceur - Casu et Macchiarulo

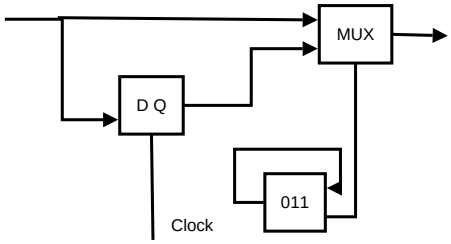


FIG. A.4 – Synchroniseur fractionnel

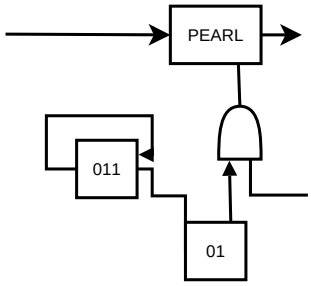


FIG. A.5 – Ordonnancement avec initialisation et partie périodique

Synchronous Latency Insensitive

Cette technique d'implantation a été introduite par Svensson dans [73]. Comme nous l'avons présenté, la latence est une métrique omniprésente dans les systèmes sur puce modernes et sera de plus en plus importante dans les générations futures. Nous avons donc besoin d'outils afin de gérer ce problème de latence/délais à deux niveaux principalement : au niveau système pour prévoir, analyser tôt les performances ; et au niveau implémentation pour disposer d'une correction vis à vis de la spécification initiale.

Au niveau système nous cherchons à partitionner la spécification en un nombre de blocs de taille limitée, de préférence une partition naturelle permettant d'isoler des "morceaux" de la fonctionnalité (par exemple processeur, mémoire ...etc...). Le système peut avoir des latences égales et fixes entre les blocs, d'une certaine manière c'est un système synchrone modulo le problème d'initialisation émergeant. D'un autre côté nous pouvons définir un système où seulement l'ordre des évènements est important comme Latency Insensitive.

Sur le plan de l'implantation nous souhaitons éviter les problèmes de synchronisation, par l'utilisation de synchronisateurs (avec le problème inhérent de la métastabilité et de latence induite), des horloges "stoppables" afin de synchroniser les données (technique héritée du GALS dans la forme originelle introduite

par Chapiro [31]), adaptation de la phase de l'horloge à la donnée, utilisation de FIFOs "élastiques" permettant d'absorber tout le biais (skew) de l'horloge et la latence résiduelle (utilisé par exemple dans la norme PCI Express sur le composant Phy(sique)).

Le flot de conception "Synchronous Latency Insensitive Design" est le suivant : partant d'une décomposition en blocs avec des régions *isochrones* (synchrones strictes sans biais), nous insérons ensuite un nombre de délais virtuels entre ces régions et nous effectuons une vérification sur les horloges, après nous remplaçons ces délais virtuels par des FIFOs élastiques qui absorbent toutes les latences du lien ainsi que le skew des horloges. L'implantation utilise en fait un port de synchronisation entre les données via l'utilisation d'un compteur recevant les entrées et un autre sur la sortie (un exemple d'implantation sur FPGA est présenté dans le papier suivant [70]). Les intérêts majeurs de cette technique d'implantation sont : l'absence totale de synchronisateur, évitant le risque de métastabilité à priori ; et aussi la possibilité de mettre en oeuvre des systèmes multi-horloges (horloges rationnelles, ici).

Approche de Singh et Theobald

Une limitation de l'approche de Carloni *et al.* est l'hypothèse que toutes les entrées doivent être lues et toutes les sorties doivent être écrites à chaque instant où le module n'est pas bloqué. Il en résulte que l'indisponibilité d'un canal d'entrée ou sortie fait que le module est bloqué, même lorsque ce canal n'est pas nécessaire pour la prochaine opération, limitant ainsi le débit du système. Entre autre, puisque le module doit produire toutes ses données nous nous retrouvons alors à convoyer des données inutiles. Une autre limitation de l'approche est de considérer seulement des liaisons point à point, qui potentiellement peuvent aussi induire une perte de performances. Finalement, le fait que nous ne considérons que les systèmes synchrones mono-horloge et non le multi-horloge, il en résulte que le débit du système est limité par le composant synchrone le plus lent.

L'impact de ces extensions est l'augmentation potentielle du débit du système, réduction de consommation électrique et une plus grande flexibilité.

Comme nous l'avons introduit précédemment, LID peut causer un ralentissement significatif du débit du système en générant plus de bloquages que nécessaire ; sur l'exemple de la figure A.6 le module $M1$ envoit une donnée tous les 10 coups d'horloge à $M2$ qui fonctionne alors pendant 9 instants successifs et $M3$ alimente $M2$ à tous les instants.

Vraisemblablement il s'agit ici d'une modélisation simple à la SDF [53] où finalement nous calculons un ordonnancement statique agissant sur l'horloge des composants ; dans SDF par contre il n'y a pas de concept de latence. LID est assimilable à un Marked Event Graph comme nous le montrerons plus loin. Néan-

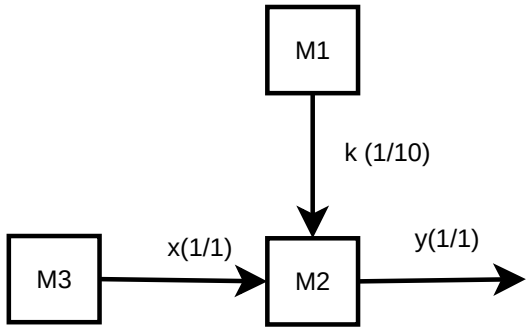


FIG. A.6 – Exemple motivant l'approche de Singh et Theobald

moins même si LID n'est pas adaptée à ce type de modélisation en terme de puissance d'expression, les remarques sont toujours valides : il est nécessaire de pouvoir gérer des situations où nous avons seulement besoin d'un sous-ensemble des entrées et des sorties. Les auteurs proposent une modification du shell qui agit sur la génération des signaux de bloquage. À la place d'effectuer la conjonction des signaux d'entrées et de sorties, nous remplaçons cette conjonction par un automate. Au lieu de générer un seul signal de bloquage, l'automate en génère jusqu'à un par entrée (sortie respectivement). L'automate permet d'implanter un ordonnanceur bien plus complexe que dans le cas de SDF, car dans ce dernier nous sommes obligés de lire sur toutes les entrées et de produire sur toutes les sorties à un rythme donné. Ainsi cette modification permet de réduire de manière significative les bloquages non nécessaires, car les bloquages ne sont plus causés par des canaux de communication inutiles lors de la réaction courante. Il en résulte une baisse significative de la puissance en évitant de transporter des informations inutiles. L'autre partie est le fait de pouvoir utiliser des topologies de communication différentes. Considérons l'exemple de la figure A.7. Nous savons

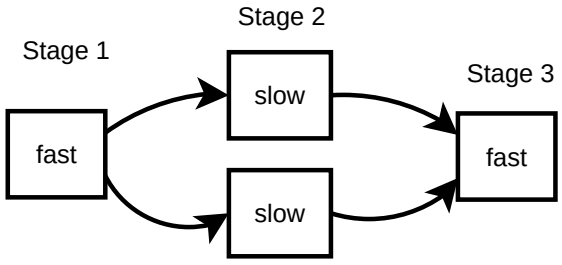


FIG. A.7 – Exemple illustrant le besoin de réseaux de communication plus complexes

que dans le système LID de Carloni *et al.*, le débit est limité par le rythme du composant synchrone le plus lent. Alors que comme l'illustre ce simple exemple où les 2 modules "slow" fonctionnent à des fréquences deux fois moindres que les autres modules, par contre il y a une autre topologie avec des cellules permettant de distribuer les données en parallèle et d'autres permettant de les rejoindre en série. Cette modification permet de s'affranchir de la contrainte précédente sur le débit qui est limité par le composant le plus lent. Lorsque le système dispose de plusieurs domaines d'horloge, alors le réseau de communication est implanté en utilisant un sous-système asynchrone permettant de lier ces différents domaines d'horloge. Même si le système n'a qu'une horloge nous pouvons aussi utiliser cette technique ou une technique synchrone. Alors le système résultant sera alors effectivement un système globalement asynchrone et localement synchrone. Ces deux dernières modifications permettent d'augmenter les performances en terme de débit et la flexibilité. Ce papier montre que l'un des principaux problèmes de performance est le modèle synchrone ; par contre il n'est en aucun cas mentionné quels sont les critères de correction d'une telle implantation candidate par rapport à quel modèle formel de spécification.

Une Implémentation Détaillée

Cette implémentation est due à Casu et Macchiarulo dans [27], il s'agissait du seul document de référence permettant de dériver une implantation opérationnelle des stations de relais et shells.

La station de relais (RS) consiste en deux registres contrôlés par un automate. La station de relais reçoit des données *DIN* avec leur signal valide *VALIN* d'une autre station de relais ou d'un shell et génère un signal de stop *STOPIN* qui est envoyé aux autres "entrées" des unités. La sortie des données est envoyée à une station de relais ou un shell qui peuvent à leur tour envoyer un signal de stop $STOPOUT$. Le premier registre est utilisé normalement pour pipeliner les données en entrée sur le front montant de l'horloge. Le second, appelé registre *auxiliaire* sauve la donnée d'entrée si $STOPOUT$ arrive lorsque nous avons une nouvelle donnée *VALIN*. L'opération est décrite graphiquement en un automate dans la figure A.8.

Après un $RESET$ l'automate entre dans l'état *processing* et y reste si $\neg STOPOUT$. Cet état est celui où normalement une donnée présente en entrée est simplement répétée. Autrement deux états peuvent être atteints depuis *processing* : *pause* si $\neg VALIN$ et *writeaux* si *VALIN*. Dans *pause*, nous restons tant que nous n'avons pas $\neg STOPOUT$ qui permet à l'automate de retourner dans l'état *processing*, ou à *writeaux* si $VALIN \wedge STOPOUT$. Dans *writeaux* la RS a une donnée valide mais ne peut l'envoyer à cause de l'évènement de stop. Ainsi la donnée est écrite dans

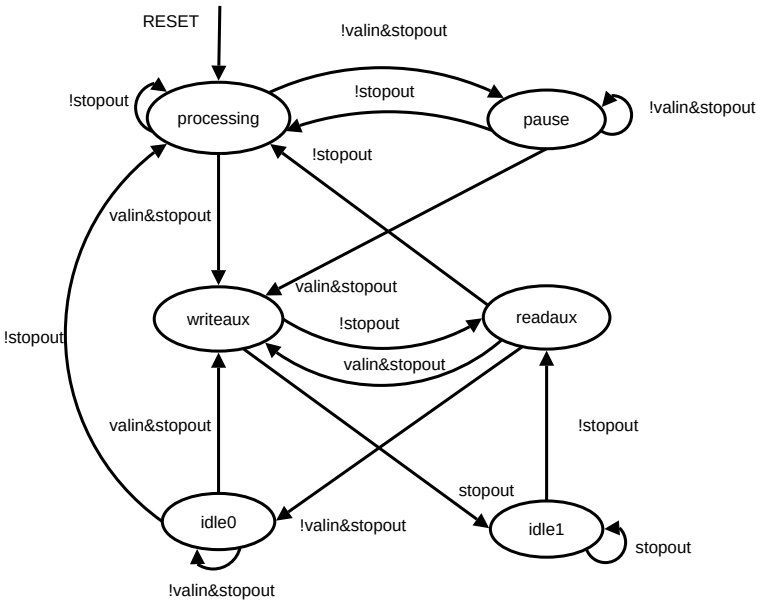


FIG. A.8 – Relay Station - Casu et Macchiarulo

le registre auxiliaire. De plus, le signal $STOPIN$ est généré et $STOPOUT$ est propagé en amont (back-pressure). Cela force la RS précédente ou le shell a conserver sa donnée jusqu'à que l'automate soit prêt à recevoir une nouvelle donnée. Si $\neg STOPOUT$ alors l'automate sort de *writeaux* et va à *readaux* où le contenu du registre auxiliaire est émis. Autrement $STOPOUT$ est présent alors l'automate va à l'état $idle1$. Dans $idle1$, le $STOPOUT$ est émis car une donnée valide en entrée doit être arrêtée. Depuis l'état précédent, la machine à états peut retourner à l'état *processing* si aucun stop n'arrive, va à *writeaux* si $STOPOUT \wedge VALIN$, à $idle0$ si $STOPOUT \wedge \neg VALIN$. Dans l'état $idle0$, la donnée de sortie précédemment lue depuis le registre auxiliaire est laissée telle quelle. Les conditions de sortie sont les mêmes que *readaux*.

Le *shell* contient la *perle*, c'est à dire le bloc fonctionnel de l'implémentation originelle, un circuit permettant d'effectuer le *clock gating*, un circuit de validation et mémorisation des données en entrée et le réseau combinatoire permettant de gérer la *back-pressure* et de générer le signal de stop. Les signaux d'entrées et de sorties sont les mêmes décrits auparavant pour la station de relais, la différence est qu'il y a ici une multitude de stations de relais qui peuvent se connecter sur les entrées et les sorties du shell. La figure A.9 illustre un shell avec deux entrées et trois sorties.

L'opération de validation est effectuée de la manière suivante : lorsqu'aucun

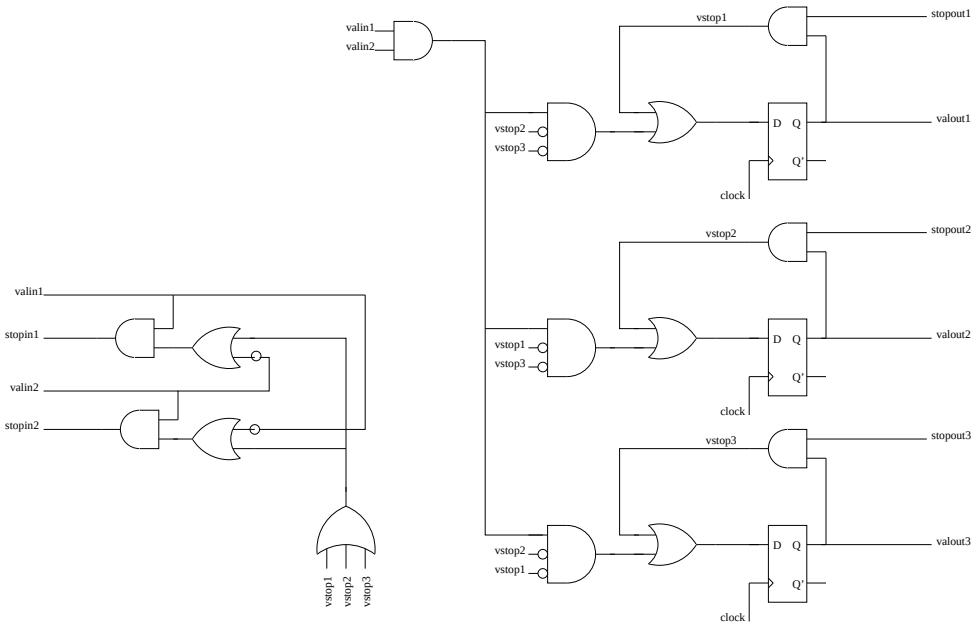


FIG. A.9 – Shell - Casu et Macchiarulo

évènement stop arrive sur les canaux de sortie, c'est à dire depuis les stations de relais connectées, les signaux $VALOUT$ sont le résultat de la conjonction de tous les signaux d'entrée $VALIN$ sur le front montant de l'horloge. Si un stop arrive sur un des canaux i c'est à dire $STOPOUT\_i = true$, et la valeur de la donnée que la station de relais doit arrêter est valide ($VALOUT_i = true$), la sortie du registre est forcée de charger un autre $true$, qui est le bit de la sortie valide stoppé sur le canal i. Si un canal i n'est pas arrêté ($STOPOUT_i = false$ ou $VALOUT_i = false$) mais le canal j l'est ($STOPOUT_j = true$ and $VALOUT_j = false$), le bit de la sortie valide est alors mise à zéro sur le canal i après le front montant de l'horloge. Les signaux $STOPOUT_i$ sont retro-propagés sur un canal d'entrée k s'ils essaient d'arrêter une donnée valide($VALOUT_i = true$) et en même temps la donnée d'entrée est valide $VALIN_k = 1$. Si la donnée d'entrée n'est pas valide cela n'a aucun sens de l'arrêter. Si elle est valide il est nécessaire de l'arrêter autrement nous perdrions la donnée car l'IP n'est pas prête.

Rôle de la Back-pressure

Cet article [20] de Carloni décrit le rôle de la back-pressure ainsi qu'une implémentation possible pour les Relay-Stations et les Shells.

La back-pressure est un mécanisme logique de contrôle de flux de l'information sur un canal de communication d'un système Latency Insensitive garantissant

qu'aucun paquet n'est perdu. La back-pressure est nécessaire pour construire des systèmes LID *"ouverts"* et peut représenter une alternative aux systèmes LID *"fermés"* car l'implémentation est modulaire et plus facile à prédire en terme de surface et puissance utilisées. Comme nous l'avons introduit précedemment les blocs de construction d'un système Latency Insensitive avec back-pressure utilisent des stations de relais et des shells : comme l'illustre la figure A.10.

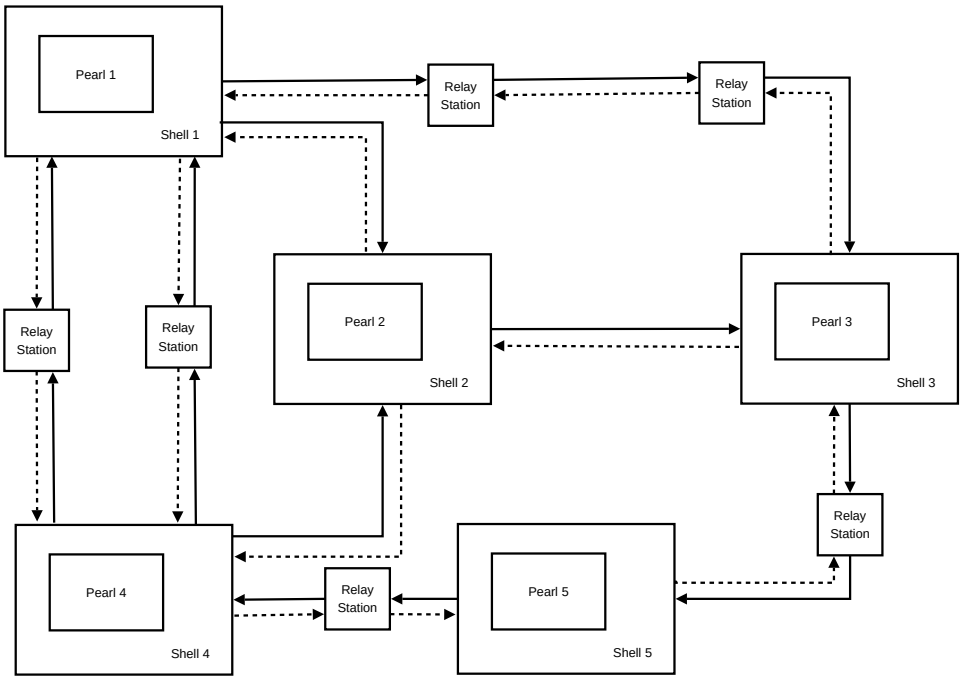


FIG. A.10 – Encaspulation avec un shell, station de relais et back-pressure

Canaux et back-pressure Les canaux dans LID sont des liaisons unidirectionnelles point-à-point qui peuvent être pipelinées comme nous l'avons décrit plus haut. Au delà de fournir le pipelining des fils, l'insertion de stations de relais sur un canal crée une sorte de "file distribuée". D'autant plus, que la logique de contrôle est également distribuée, il en résulte une structure modulaire. À cause de la nature de plus en plus distribuée sur les systèmes sur puce, des files distribuées représentent une solution de design plus prometteuse que disposer de longues files de communications centralisées placées à côté de chaque IP (mais les contraintes de placement/routage peuvent nécessiter de placer ces files près de l'IP).

Implémentation Relay-Station Une implémentation possible pour la relay-station est donnée dans la figure A.13 sous la forme d'un circuit synchrone : à chaque

coup d'horloge t la relay-station prend un paquet et un flag de stop en entrée et émet un paquet en sortie avec un flag de stop. Les signaux de stop constituent le mécanisme de back-pressure et le contrôle de flot est assuré par les signaux paquets. Une propriété clef de la relay-station est qu'il n'existe pas de boucle combinatoire entre ses entrées et sorties. Il faut donc un cycle d'horloge pour propager les différents signaux. Puisque nous devons ne pas perdre de donnée : nous ne devons pas oublier l'ensemble des données que nous recevions en amont alors qu'en aval nous recueillons en même temps un stop, il faut alors disposer d'un espace suffisant pour fonctionner au débit maximum en utilisant 2 places (un registre *Main* et un autre *Aux*), nous reviendrons sur ce point ultérieurement.

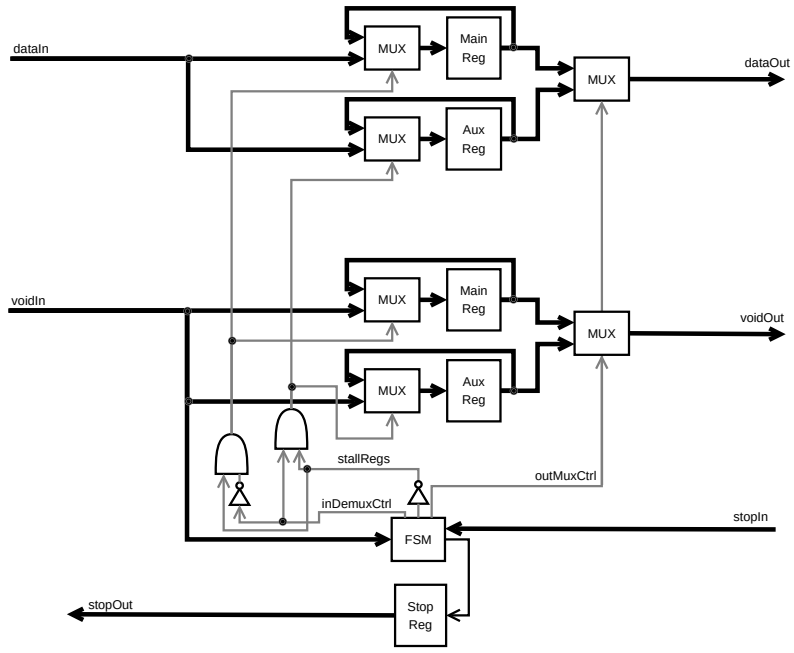


FIG. A.11 – Implémentation RS RTL - Carloni

La figure A.13 contient la machine à état décrivant le contrôle de la relay-station. Cet automate a 2 signaux d'entrée (*stopIn*, *voidIn*), 3 signaux de sortie (*inDemuxCtrl*,*outMuxCtrl*,*stallRegs*), et 4 états (*Processing*, $WriteAux$, $Stalling$, *ReadAux*). Les signaux *outMuxCtrl*, *stallRegs* dépendent seulement sur l'état courant de l'automate alors que le signal *inDemuxCtrl* dépend à la fois de l'état courant et du signal d'entrée *stopIn*. L'automate ne contient pas la logique pour gérer la valeur de $stopOut^{t+1}$ qui est simplement égale à la valeur de $stopIn_t$ à moins que la station de relais soit dans l'état *Processing* et $voidIn^t = 1$: dans ce cas $stopOut^{t+1}$ est égal à 0. Notons aussi la caractéristique suivante du protocole insensible à la latence implanté par cette station de relais : si le signal *stopIn* reste

à 1 dans un module en aval (station de relais ou shell) pour un seul cycle d'horloge, alors la station de relais n'a pas vraiment besoin de bloquer (aucun paquet n'est conservé sur le port de sortie pour plus d'un seul cycle). Réciproquement, la station de relais sait que le module en aval est capable de gérer le paquet courant sur sa sortie à un temps t donné lorsque la condition suivante est satisfaite : $(stopIn^t = 0) \vee (stopIn^t \wedge stopIn^{t-1} = 0)$

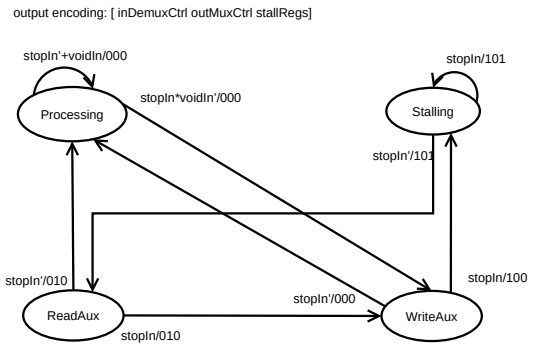


FIG. A.12 – Implémentation RS FSM - Carloni

Encapsulation par un Shell Étant donné une IP M, une instance d'un shell peut être automatiquement synthétisée afin d'encapsuler M et l'interfacer avec des canaux de façon à ce que le système soit "patient". La théorie du LID garantit que la seule précondition nécessaire est que M soit "bloquable". La première tâche du shell est d'assurer la synchronisation des données, via le protocole LID ; la seconde est de générer l'horloge. Lorsque le shell doit bloquer, ce dernier doit stocker les données présentes sur l'entrée et propager des sorties. Sinon lorsque que toutes ses entrées sont prêtes et qu'il n'y a pas de back-pressure active alors l'IP est exécutée, en suivant une règle d'exécution ASAP (As Soon As Possible).

La logique de contrôle implémente les différentes tâches décrites auparavant. Le mécanisme de bloquage/exécution est simplement obtenu avec l'utilisation du "clock-gating", qui contrôle les registres de l'IP. Notons que toutes les sorties sont "latchées" : les données par l'IP, et les signaux de contrôle *void* et $stop$ par le shell lui-même. Finalement notons aussi les files court-circuitables sur les canaux d'entrées. La raison pour les rendre court-circuitables est de garantir que lorsqu'un nouvel ensemble de paquets sont disponibles et qu'il n'y a aucune requête de bloquage arrivant des canaux de sortie alors ce nouvel ensemble est produit exactement en un cycle d'horloge, c'est à dire sans ajouter un cycle à la latence originelle de l'IP.

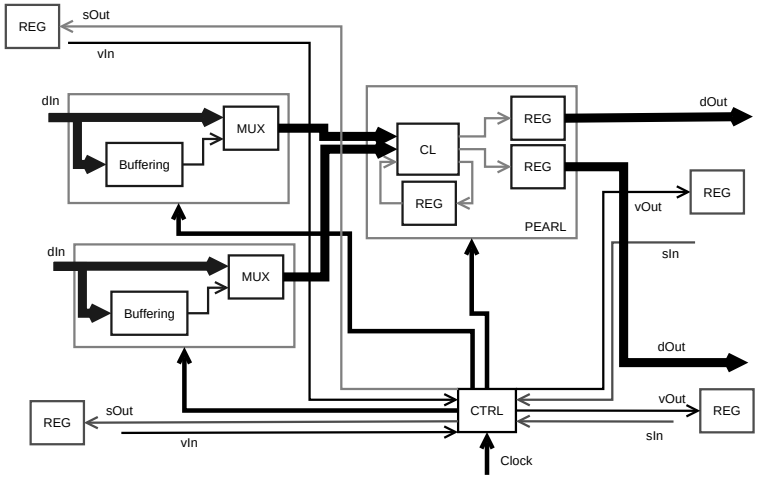


FIG. A.13 – Implémentation Shell RTL - Carloni

Performances de LID Quelque soit le nombre de stations de relais introduites sur les canaux insensibles à la latence, sa correction fonctionnelle est assurée : le système peut produire plus de paquets "vides" sur les canaux de sortie ainsi que plus de back-pressure sur les canaux d'entrée, mais sans pour autant "deadlocker". Le problème d'efficacité dans le cas de LID est d'assurer des performances suffisantes avec des latences augmentées sur les canaux de communication. Afin d'évaluer correctement les performances d'un système latency-insensitive, il est nécessaire de vérifier à quelle fréquence il produit de la back-pressure et des paquets vides. Ainsi, le débit d'un système dépend du nombre de paquets "utiles" à un intervalle de temps donné. Cela correspond au rapport entre le nombre de paquets utiles sur la somme des paquets vides et utiles.

L'origine des paquets vides : un système latency insensitive peut recevoir des paquets vides depuis son environnement ou peut les générer lui-même. Dans le premier cas, c'est l'environnement qui impacte sur le débit. Le second cas est plus intéressant d'un point de vue design car il pose une limite sur le débit maximum que le système est capable d'atteindre. Un shell correctement conçu émet des paquets vides sur ses canaux de sortie seulement s'il est forcé de bloquer. La structure du LID détermine le débit maximum atteignable, si le système est acyclique son débit est maximal s'il est cyclique il est borné par le circuit le plus lent en débit, le débit est le ratio entre le nombre de valeurs initiales (jetons) et la somme des latences du dit-circuit.

Modélisation de systèmes insensibles à la latence avec des Marked Event Graphs : la structure et le comportement de l'implémentation peuvent être décrits

à l'aide d'un Marked Event Graph. L'approche est constructive dans la mesure où il y a une bijection entre les composants du LID et des constructions des Marked Event Graphs. Nous observons que l'implémentation "originale" du LID peut être modélisée au niveau du protocole dont le comportement est entièrement déterminé une fois que les paquets vides ont été distingués des vrais paquets dans l'état initial.

Cette méthode constructive basée sur les Marked Event Graphs permet d'analyser les implantations "virtuelles" où nous avons placé des places non-bornées ainsi que les implantations avec des places bornées. Dans ces deux cas, nous pouvons utiliser les propriétés théoriques des Marked Event Graphs pour prouver si le système LID est vivace par construction ; déterminer si ou non il est borné, et de calculer statiquement le débit maximum atteignable. Le modèle avec les capacités infinies peut être utilisé afin de déterminer la taille minimale finie afin que le système fonctionne au débit maximal. Ainsi, nous savons que nous pouvons construire une implémentation physique basée sur la back-pressure et de taille finie avec une longueur égale à 2 qui offre les même performances qu'une implémentation virtuelle.

Discussion Dans ce paragraphe nous fournissons une analyse qualitative de certains aspects clefs qui devront être implantés dans un système insensible à la latence.

Le rôle de la topologie du système Comme discuté auparavant, le débit maximum atteignable et ultimement la performance du système insensible à la latence dépend d'où et combien de stations de relais ont été insérées. Ainsi, il nous suffit d'analyser directement la structure du système correspondante au Marked Event Graph. Il y a quatre cas :

- Le Marked Event Graph est acyclique : l'insertion de stations de relais sur n'importe quel canal du système considéré n'a aucune incidence sur son débit. Quelques paquets vides sont observés sur les sorties du système, forçant les vrais paquets correspondants à arriver avec quelques cycles d'horloge de latence additionels, il peut arriver éventuellement que le système atteigne un régime périodique où aucun paquet vide est observé. Entre autre, aucune accumulation de jeton non-bornée ne peut arriver dans un tel système. Des files de longueur finies sont nécessaires dans ces shells qui ont de multiples canaux d'entrée et au moins l'un d'entre eux reçoit certains paquets vides. Ces canaux peuvent être identifiés grâce aux emplacements des stations de relais et de la topologie du système.
- Le Marked Event Graph est un ensemble de parties fortement connexes reliées via des stations de relais : l'insertion de stations de relais entre les

parties fortement connexe n'altère en aucun cas le débit de ces dernières ; par contre le débit du système est borné par le cycle de débit minimal.
- Le Marked Event Graph est une seule partie fortement connexe : l'insertion de station de relais sur n'importe quel canal de communication a toujours un impact négatif sur le débit maximum atteignable. L'impact varie en fonction de la structure des cycles du système et peut être calculé précisemment en calculant le débit du système. Il n'y a pas d'accumulation non-bornée de jetons à l'intérieur du système que l'implantation soit basée sur l'utilisation de la back-pressure ou non. Cependant il peut y avoir une accumulation non-bornée de jetons entre le système et l'environnement avec lequel il opère.
- Le Marked Event Graph est un ensemble de parties fortement connexes reliées en interne/externe avec des stations de relais : pour chaque partie fortement connexe nous nous ramènons au cas précédent, par contre entre les parties fortement connexes il peut y avoir une accumulation non bornée de jetons il est nécessaire de ralentir les parties fortement connexes les plus lentes, ou d'utiliser de la back-pressure.

Le rôle de la back-pressure Le choix d'utiliser ou non de la back-pressure au niveau de l'implémentation physique n'est pas évident. Nous savons déjà que ce choix n'affectera pas le débit maximum atteignable. En d'autres termes, tant que l'environnement est capable d'envoyer de vrais paquets et ne génère pas pour toujours des requêtes de bloquage, la back-pressure n'est pas le facteur décisif pour déterminer les performances du système. D'un autre côté, la back-pressure nécessaire pour garantir que n'importe quelle spécification puisse être implémentée quelque soit le nombre de stations de relais. La back-pressure suppose que l'environnement est prêt à bloquer lorsque le système envoie un signal de bloquage. De ce point de vue, pour une implémentation physique nous disposons de deux alternatives pour éviter l'accumulation de jetons :
- utiliser la back-pressure et réduire dynamiquement le taux de production des jetons de l'environnement de façon à ce qu'il corresponde avec le débit maximum du système insensible à la latence.
- ne pas utiliser la back-pressure et réduire statiquement le taux de production des jetons de l'environnement de telle manière à ce que l'horloge de ce dernier et l'horloge effective du système insensible à la latence correspondent.

Ce choix peut être restreint lorsque le système est "ouvert", c'est à dire un système qui doit être conçu sans être capable de contrôler le design des autres systèmes qui constituent son environnement. Dans ce cas, si l'environnement est lui-même insensible à la latence alors la back-pressure doit être utilisée ; s'il n'est pas insensible à la latence, alors le design final sera correct pour les environnements fonctionnant à la même horloge effective du système considéré.

Lorsqu'un système est "fermé", par contre la décision précédente est influencée par des considérations de puissance/surface puisque les performances sont identiques. Dans le cas de l'utilisation de la back-pressure nous disposons d'une technique modulaire, entre autre le coût en surface engendré par l'ajout des shells et stations de relais est facile à calculer. Il faut par contre prendre garde aux fils supplémentaires ajoutés. L'alternative est de ne pas utiliser de back-pressure, nous enlevons ainsi les fils précédents, nous simplifions les stations de relais par des répéteurs, par contre il est nécessaire d'égaliser les débits des parties fortement connexes pour éviter les accumulations de jetons. Voici une autre raison pourquoi il est nécessaire d'avoir une design "équilibré", un design où les latences de communication et de calcul sont bien équilibrées.

Approche de Suhaib *et al.*

Dans [72], les auteurs introduisent une autre technique d'implémentation de LID et utilisent une stratégie de vérification formelle pour être sûr qu'au fur et à mesure que le protocole est rafiné, il ne devienne pas inéquivalent d'un point de vue fonctionnel à la spécification de haut niveau synchrone.

D'abord ils montrent comment se passer des stations de relais, ensuite ils détaillent une extension des interfaces afin de permettre de disposer d'horloges rationnelles ; enfin ils utilisent une technique de vérification formelle afin de maintenir le lien d'équivalence avec le design latency insensitive. Ils proposent de résoudre le problème de latence sans utiliser de stations de relais. L'idée de l'approche est que si un évènement entre deux composants prend deux cycles d'horloge, nous envoyons seulement une valeur un cycle sur deux adaptant ainsi la vitesse de communication entre les composants par leur distance. Une interface gère cette restriction et bloque le composant émetteur qui délivre ses valeurs trop vite. Même si les stations de relais sont éliminées cela ralentit très fortement le système à cause du nombre de bloquages qui augmente. Ils éliminent ce problème en rajoutant des lignes de communication additionelles. Le nombre d'interconnections dépend du délais d'interconnection. Pour un délais de n cycles il faut donc n interconnections placées entre les IPs. Ces fils additionnels ajoutent de la surface et des difficultés dans le routage ; l'argument des auteurs est que dans les processus modernes cela n'est pas très coûteux. Le deuxième argument est qu'il est rare d'avoir des designs où les longs fils font plus de 2 ou 3 cycles d'horloge, auquel cas nous n'utiliserions pas le LID. Pour s'assurer que les évènements sont correctement transférés d'une IP à une autre ils ajoutent un *spliter* sur la source et un *merger* sur la destination avec des interconnections additionelles. Après les auteurs montrent avec une preuve par rafinement que c'est équivalent au système LID usuel. Plus tard ils font une extention vers le multi-clock où l'idée est de rafiner le wrapper d'égalisation (équivalent à l'étage de bufferisation du *Shell*) en

charge de synchroniser l'arrivée des différentes données via les interconnections, ici multiclock signifie que les composants ont des fréquences rationelles, il existe donc une horloge de base globale bien plus rapide. Au plus une valeur est donnée en lecture dans l'égaliseur, alors l'IP est évaluée et produit une valeur récupérée par le splitter/merger qui lit/écrit alors la valeur, le splitter/merger fonctionne sur l'horloge de base qui est la plus rapide, nous écrivons un τ à chaque fois qu'il n'y a pas de valeur. Le papier [71] montre comment vérifier ces précédents modèles en utilisant un environnement de programmation fonctionnel (Standard ML) permettant de modéliser, implémenter la sémantique des différents rafinements de façon à valider le modèle par rapport à la spécification originelle en utilisant l'équivalence modulo latence.

A.1.5 Résumé

Dans cette section nous avons introduit le modèle des systèmes dits-insensibles à la latence. Étant donné une hypothèse que nous qualifions de *"patience"*, c'est à dire que le fait d'appliquer du clock-gating ne modifie pas le comportement du système, il est ainsi possible de partir d'une spécification synchrone où à cause de contraintes de placement/routage il existe de "longs" fils où il n'est pas possible d'atteindre la fréquence d'horloge requise. La solution proposée est alors d'encapsuler chaque sous-système avec un wrapper appelé *Shell* et de disposer un certain nombre de *Stations de Relais* de façon à ce qu'il n'y ait plus que des fils dont le délais soit inférieur ou égal à un cycle d'horloge. Il en résulte alors que nous avons effectué une déformation temporelle de l'exécution de la spécification synchrone sur un nombre borné de cycle d'horloges. Nous avons introduit de manière détaillée le modèle initial théorique originel et rappelé l'ensemble des lemmes et théorèmes montrant la correction d'un tel système par rapport à spécification via l'équivalence modulo latence, qui est intuitivement que d'un ordre partiel d'évènements nous obtenons un autre ordre partiel compatible avec le premier. Puis nous avons étendus sur les travaux les plus récents attaché à ce modèle :

- l'approche *recycling* de Carloni où le but est de rajouter des *stations de relais* de façon à limiter l'utilisation du protocole de *back-pressure* sans altérer les performances (dans ce cadre le débit) ;
- une approche inspirée du *software-pipelining* de Casu et Macchiarulo où cette fois ci les auteurs ont cherché à supprimer complètement le protocole grâce à un ordonnancement statique permettant de conserver le débit maximum du système (*NB :* ce système n'est pas compositionel tel quel) ;
- le *synchronous latency insensitive* de Svensson où nous utilisons des FIFOs *"élastiques"* afin d'absorber les latences résiduelles entre différents chemins ne disposant pas des mêmes latences ;
- l'approche de Singh et Theobald où ils discutent comment "relâcher" le modèle en autorisant lorsque c'est possible à laisser le module s'éxécuter lorsqu'il a qu'un sous-ensemble de ses entrées, et à permettre aussi la mise en œuvre de systèmes multi-horloge au lieu du synchrone ;
- une implémentation détaillée (niveau RTL) des stations de relais et du shell due à Casu et Macchiarulo, ce travail permet de disposer d'une vue opérationnelle du fonctionnement du LID, et est le point de départ d'une des contributions ;
- le rôle de la back-pressure dans le LID qui discute de différents problèmes du LID en terme de compositionalité et de son lien avec le modèle des Marked Event Graphs ; nous étendrons de manière plus détaillée ce lien effectif dans les contributions ;

– une autre technique d'implantation due à Suhaib *et al.* supprimant les stations de relais en modifiant les liaisons d'interconnections afin d'augmenter leur débit proportionellement à la latence et utilisant deux nœuds *splitter/merger*, la synchronisation est effectuée au niveau du *shell*.

Il existe aussi une variante asynchrone de LID appelée *SELF* [50, 35, 48] et dont l'implantation compositionnelle est plus efficace en terme de surface que le LID, les résultats théoriques sont de la même nature.

Annexe B

SDF

Ce chapitre présente des preuves issus du papier original d'Edward Lee.

Lemme 27. *Toutes les matrices topologiques pour un graphe SDF donné ont le même rang.*

Démonstration. Les matrices topologiques sont apparentées par renumérotation des nœuds et arcs, cela se traduit par des permutations de lignes et colonnes dans la matrice topologique. De telles opérations préservent le rang. □

Lemme 28. *Une matrice topologique pour un arbre à un rang $s-1$ où s est le nombre de nœuds (un arbre est un graphe connecté sans cycle, où nous ignorons la direction des arcs).*

Démonstration. Preuve par induction. Le lemme est clairement vrai pour un arbre avec deux nœuds. Supposons que pour un arbre avec N nœuds $rang(\tau) = N-1$ soit vrai. Ajoutons un nœud et un lien connectant ce nœud à notre graphe qui sera alors un arbre avec $N+1$ nœuds. Une matrice topologique pour ce nouveau graphe peut être écrite $\tau_{N+1} = [\frac{\tau_N | O}{\rho^\tau}]$ où O est un vecteur colonne rempli de zéros, et ρ^τ est un vecteur ligne correspondant à l'arc que nous venons juste d'ajouter. La dernière entrée dans le vecteur ρ^τ est non-nulle car le nœud que vous ajouté auparavant correspond à la dernière colonne, et doit être connecté au graphe. Ainsi, la dernière ligne est linéairement indépendante des autres lignes, donc $rang(\tau_{N+1}) = rang(\tau_N) + 1$. □

Lemme 29. *Pour un graphe SDF connexe avec une matrice topologique τ : $rang(\tau) \geq s-1$ où s est le nombre de nœuds dans le graphe.*

Démonstration. Considérons n'importe quel arbre recouvrant τ du graphe SDF connexe (un arbre recouvrant est un arbre incluant chaque nœud du graphe). Maintenant définissons τ_T, la matrice topologique de ce sous-graphe. Par le précédent

lemme, nous avons $rang(\tau_T) = s-1$. Ajouter des arcs à ce sous-graphe revient à simplement rajouter des lignes à la matrice topologique. Ajouter des lignes dans une matrice peut augmenter le rang, si les lignes sont linéairement indépendantes des lignes existantes, mais il ne peut être diminué. □

Corollaire 30. *Le rang d'une matrice topologique est $s-1$ ou s.*

Démonstration. τ a seulement s colonnes, le rang ne peut excéder s. □

Un *ordonnancement séquentiel admissible* ϕ est une liste ordonnée non-vide de nœuds telle que si les nœuds sont exécutés dans la séquence donnée par ϕ, la quantité de données dans les buffers sera non-négative et bornée. Chaque nœud doit apparaître dans ϕ au moins une fois. Un *ordonnancement séquentiel périodique admissible* (Periodic Admissible Sequential Schedule - PASS) est un ordonnancement séquentiel périodique et infini admissible. Il est spécifié par une liste ϕ qui est la liste des nœuds dans une période.

Théorème 31. *Pour un graphe connecté SDF avec s nœuds et une matrice topologique τ, $rang(\tau) = s-1$ est une condition nécessaire pour qu'un ordonnancement séquentiel périodique admissible existe.*

Démonstration. Nous devons montrer que l'existence d'un PASS de période p implique $rang(\tau) = s-1$.

En utilisant l'équation suivante modélisant le changement de taille de la file lors d'une exécution d'un nœud : $b(n+1) = b(n) + \tau v(n)$ où $v(n)$ est un vecteur colonne décrivant un ordonnancement séquentiel ($v(n) \in [0;1]$ où 1 est une exécution). Nous pouvons alors écrire :
$b(p) = b(0) + \tau_q$ où $q = \Sigma_{n=0}^{p-1} v(n)$.

Puisque l'ordonnancement est périodique nous pouvons écrire :
$b(np) = b(0) + n\tau_q$.

Puisque l'ordonnancement est admissible, les buffers doivent être bornés. Ils sont bornés si et seulement si $\tau q = O$ où O est un vecteur rempli de zéros. Pour $q \neq O$, cela implique que $rang(\tau) < s$ où s est la dimension de q. Ainsi en utilisant la précédente corollaire, alors nous obtenons $rang(\tau) = s-1$. □

Ainsi si le rang de notre graphe SDF a une matrice de $rang(\tau) = s$, alors il n'existe pas d'ordonnancement avec des buffers bornés et le système arrivera tôt ou tard sur un blocage.

Lemme 32. *Supposons un graphe SDF connecté avec une matrice topologique* τ. *Soit q n'importe quel vecteur tel que* $\tau q = O$. *Prenons un chemin connecté passant au travers du graphe par l'ensemble* $B = \{b_1, \ldots, b_L\}$ *où chaque entrée désigne un nœud, et le nœud* b_1 *est connecté au nœud* b_2*,..., jusqu'au nœud* b_L. *Alors tous les* q_i, $i \in B$ *sont des zéros, ou sont tous strictement positifs, ou sont tous strictement négatifs. De plus, si au moins un* q_i *est rationnel alors tous les* q_i *sont rationnels.*

Démonstration. Par induction. Tout d'abord considérons un chemin entre deux nœuds, $B_2 = \{b_1, b_2\}$. Si l'arc connectant ces nœuds est le j^{ieme} arc, alors : $q_{b_1}\tau_{jb_1} + q_{b_2}\tau_{jb_2} = 0$ (par définition de la matrice topologique, la j^{ieme} ligne a seulement deux entrées). Aussi par définition, τ_{jb_1} et τ_{jb_2} sont des entiers non-nuls de signe opposé. Le lemme s'applique immédiatement pour B_2.

Maintenant, supposons que le lemme est vrai pour B_n, prouvons que cela est vrai pour B_{n+1} est trivial, en utilisant le même raisonnement comme dans le cas précédent, et en considérant la connection entre les nœuds b_n et b_{n+1}.

□

Théorème 33. *Pour un graphe SDF connexe avec s nœuds et une matrice topologique* τ, *avec* $rang(\tau) = s - 1$, *nous pouvons trouver un vecteur d'entiers positifs* $q \neq O$ *tel que* $\tau q = O$ *où O est le vecteur nul.*

Démonstration. Puisque $rang(\tau) = s - 1$, un vecteur $v \neq O$ peut être trouvé tel que $\tau v = O$. De plus, pour n'importe quel scalaire α, $\tau(\alpha v) = O$. Soit $\alpha = 1/v_1$ et $v' = \alpha v$. Alors $v'_1 = 1$, et par la corollaire du précédent lemme, tous les autres éléments dans v' sont des nombres rationnels positifs. Soit c un multiple commun de tous les dénominateurs des éléments de v' et soit $q = cv'$. Alors q est un vecteur d'entiers positifs tel que $\tau q = O$. □

Il est intéressant de résoudre pour le plus petit vecteur d'entiers positifs. Pour faire cela, il faut réduire chaque rationnel dans v' en utilisant par exemple l'algorithme d'Euclide. Pareillement, pour l'obtention du plus petit commun multiple entre les dénominateurs. Il en résultera que cv' sera le plus petit vecteur d'entiers positifs.

Nous avons maintenant une condition nécessaire pour l'existence d'un ordonnancement admissible, que le rang de la matrice topologique soit de $s - 1$. Afin de disposer d'une condition suffisante pour trouver un tel ordonnancement nous allons caractériser une classe d'algorithmes qui trouveront un PASS s'il existe. L'utilisation d'un tel algorithme est une condition suffisante pour montrer l'existence d'un ordonnancement admissible.

Définition 46. Un prédecesseur d'un nœud x est un nœud générant des données pour x.

Lemme 34. *Pour déterminer si un nœud x d'un graphe SDF peut être ordonnancé au temps i, il est suffisant de connaître combien de fois x et ses prédecesseurs ont été ordonnancés, et de connaître $b(0)$, qui est l'état initial des buffers. C'est à dire que nous n'avons pas besoin de savoir dans quel ordre les prédecesseurs ont été ordonnancés ni si d'autres nœuds ont été ordonnancés entre temps.*

Démonstration. Pour ordonnancer un nœud n, chaque buffer en entrée doit disposer de suffisament de données. La taille de chaque buffer j au temps i est donnés par $[b(i)]_j$, la j^{ieme} entrée dans le vecteur $b(i)$. Nous pouvons écrire $b(i) = b(0) + \tau q(i)$ où $q(i) = \Sigma_{n=0}^{i-1} v(n)$. Le vecteur $q(i)$ contient seulement les informations à propos de combien de fois chaque nœud a été invoqué avant l'itération i. Les tailles de buffer $[b(i)]_j$ dépendent seulement de $[b(0)]_j$ et $[\tau q(i)]_j$. La j^{ieme} ligne de τ a seulement deux entrées, correspondantes aux deux nœuds connectés sur le j^{ieme} buffer, ainsi seulement les deux entrées correspondantes du vecteur $q(i)$ peuvent affecter la taille de buffer. Ces entrées spécifient le nombre de fois que ξ et ses prédecesseurs ont été invoqués, ainsi cette information et les tailles initiales des buffers $[b(0)]_j$ est tout ce dont nous avons besoin. □

Définition 47 (Algorithmes de classe S)**.** Étant donné un vecteur d'entiers positifs q tel que $\tau q = O$ et un état initial des buffers $b(0)$, le i^{eme} nœud est *exécutable* à un temps donné s'il a été exécuté q_i fois et l'exécuter n'engendrera pas une taille de buffer négative. Un *algorithme de classe S* est n'importe quel algorithme qui ordonnance un nœud s'il est exécutable, mettra à jour $b(n)$ et s'arrêtera (terminera) seulement lorsqu'aucun nœud sera exécutable. Si un algorithme de classe S termine avant qu'il ait ordonnancé chaque nœud le nombre de fois spécifié dans le vecteur q, alors il sera dit bloqué.

Les algorithmes de la classe S (pour Séquentiel) construisent des ordonnancements statique par simulation des effets sur les buffers lors d'une exécution. C'est à dire, que les nœuds "programmes" ne sont pas rellement exécutés en fait. Ainsi, n'importe quel ordonnancement dynamique devient un algorithme de classe S simplement en spécifiant une condition d'arrêt, qui dépend du vecteur q. Il est nécessaire de prouver que la condition d'arrêt est suffisante pour construire un *PASS* pour n'importe quel graphe valide.

Théorème 35. *Étant donné un graphe SDF avec une matrice topologique τ et étant donné un vecteur d'entiers positifs q tel que $\tau q = O$, si un PASS de période $p = 1^T q$ existe, où 1^T est un vecteur ligne rempli de* 1*, n'importe quel algorithme de classe S trouvera un tel PASS.*

Démonstration. Il est suffisant de montrer que si un PASS ϕ existe de n'importe quelle période p, un algorithme de classe S ne bloquera pas avant que la condition de terminaison soit satisfaite.

Supposons qu'un PASS ϕ existe, et définissons $\phi(n)$ comme ses n entrées, pour n'importe quel n tel que $1 \leq n \leq p$. Supposons un algorithme donné de la classe S qui construit itérativement un ordonnancement, et définissons $\xi(n)$ la liste des premiers nœuds ordonnancés par l'itération n.

Nous avons besoin de montrer que lorsque n augmente, l'algorithme construira $\xi(n)$ et ne bloquera pas avant $n = p$, lorsque la condition de terminaison sera satisfaite. C'est à dire, nous avons besoin de montrer que pour tout $n \in (1, ..., p)$, il y un nœud qui est exécutable pour n'importe quel $\xi(n)$ que l'algorithme peut avoir construit.

Si $\xi(n)$ est n'importe quelle permutation de $\phi(n)$ alors la $(n+1)^{ieme}$ entrée dans ϕ est exécutable grâce au précédente lemme car tous les prédecesseurs nécessaire doivent être dans $\phi(n)$, et donc dans $\xi(n)$. Autrement, le premier nœud α dans $\phi(n)$ est exécutable et non pas dans $\xi(n)$, aussi grâce au précédent lemme. Ceci est vrai pour tout $n \in (1, ..., p)$, ainsi l'algorithme ne bloquera pas avant $n = p$.

À $n = p$, chaque nœud i a été ordonnancé q_i fois car chaque nœud ne peut être ordonnancé plus de q_i fois et $p = 1^T q$. Ainsi la condition de terminaison est satisfaite et $\xi(p)$ est un ordonnancement admissible.

□

Le précédent théorème nous dit qu'il existe un vecteur d'entiers positifs q dans "l'espace vide" de la matrice topologique, les algorithmes de la classe S trouveront un ordonnancement avec une période égale à la somme des éléments dans le vecteur, si un PASS existe. Il est possible même si $rang(\tau) = s - 1$ qu'aucun PASS n'existe, à cause du nombre de données placées initialement.

Il y a un autre problème. Il y a un nombre infini de vecteurs dans cet espace vide pour la matrice topologique. Comment en sélectionner un à utiliser dans les algorithmes de la classe S ? Maintenant, nous allons prouver que quel que soit l'algorithme de la classe S, si l'un d'entre eux n'arrive pas à trouver de PASS alors aucun PASS n'existe quelle que soit la période considérée.

Lemme 36. *Connecter un nœud de plus au graphe augmente le rang de la matrice topologique d'au moins un.*

La preuve de ce lemme suit les mêmes types d'arguments que dans le lemme 28. Des lignes sont ajoutées à la matrice topologique afin de décrire les connections ajoutées au nouveau nœud, et ces lignes doivent être linéairement indépendantes des lignes déja existantes dans la matrice topologique.

Lemme 37. *Pour n'importe quel graphe SDF connexe avec s nœuds et une matrice topologique* τ*, un sous-graphe connexe L avec m nœuds a une matrice topologique* τ_L *pour laquel :* $rang(\tau) = s-1 \rightarrow rang(\tau_L) = m-1$*, c'est à dire tous les sous-graphes ont le même rang.*

Démonstration. Preuve par l'absurde. Nous montrons que $rang(\tau_L) \neq m-1 \rightarrow rang(\tau) \neq s-1$. En utilisant la corollaire du Lemme 29, si $rang(\tau_L) \neq m-1$ alors $rang(\tau_L) = m$. Alors $rang(\tau) \geq m+(s-m) = s$, par l'application répétée du précédent lemme, ainsi $rang(\tau) = s$. □

Le prochain lemme montre qu'étant donné un vecteur de l'espace vide de q, afin d'exécuter n'importe quel nœud le nombre de fois spécifié par ce vecteur, il n'est pas nécessaire d'exécuter n'importe quel autre nœud plus que le nombre de fois spécifié par le vecteur.

Lemme 38. *Considérons le sous-graphe d'un graphe SDF formé par n'importe quel nœud* α *et tous ses prédecesseurs immédiats (nœuds qui envoient des données, qui peut aussi inclure* α *lui-même). Construisons une matrice topologique* τ *pour ce sous-graphe. Si le graphe originel a un PASS, alors par le théorème 31 et le lemme 37, alors* $rang(\tau) = m-1$ *où m est le nombre de nœuds dans le sous-graphe. Trouver n'importe quel vecteur d'entiers positifs q tel que* $\tau q = O$*. Un tel vecteur existe à cause du théorème 33. Alors il n'est jamais nécessaire de lancer n'importe quel prédecesseur* β *plus de* q_b *fois de façon à exécuter x fois, pour n'importe quel* $x \leq q_\alpha$*.*

Démonstration. Le nœud α ne consommera aucune donnée produite par la y^{ieme} execution de β pour n'importe quel $y > q_b$. De la définition de τ et q nous savons que $aq_a = bq_b$ où a et b sont le nombre de données consommées et produites par le lien de β à α. Ainsi, exécuter β seulement q_β fois génère suffisamment de données sur le lien pour exécuter q_α fois. Exécuter plus de fois n'aidera pas. □

Théorème 39. *Étant donné un graphe SDF avec une matrice topologique* τ *et un vecteur d'entiers positifs q tel que* $\tau q = O$*, un PASS de période* $p = 1^T q$ *existe si et seulement si une période Np existe pour n'importe quel entier N.*

Démonstration. Première partie : il est trivial de montrer l'existence d'un PASS de période p implique l'existence d'un PASS de période Np, car le premier PASS peut être appliqué N fois pour produire le second.

Seconde partie : nous montrons maintenant l'existence d'un PASS ϕ de période Np implique l'existence d'un PASS de période p. Considérons le sous-ensemble δ de ϕ alors il s'agit d'un ordonnancement de période p et c'est finis. Considérons alors que ce ne soit pas le cas, alors il existe un nœud β qui a été

exécuté plus de q_β fois avant tous les nœuds qui ont été exécutés q fois. Mais par le lemme 38, le fait "qu'il y ait plus d'exécutions que q" de β n'est pas nécessairement pour le "qu'il y ait moins ou autant d'exécutions que q" qui peut arriver plus tard. Ainsi le "qu'il y ait moins ou autant d'exécutions que q" peut être déplacé dans le haut de la liste ϕ de telle manière à ce qu'ils précèdent tous les "qu'il y ait plus d'exécutions que q" de β, produisant un nouveau PASS ϕ' de période Np. Si le processus est répété jusqu'à ce que toutes les "qu'il y ait moins ou autant d'exécutions que q" précèdent tous les "qu'il y ait plus d'exécutions que q", alors les p premiers éléments de l'ordonnancement résultant constituerons un ordonnancement de période p. □

Corollaire 40. *Étant donné n'importe quel vecteur d'entiers positifs $q \in \nu(\tau)$, l'espace vide de τ, un PASS de période $p = 1^T q$ existe si et seulement si un PASS existe de période $r = 1^T v$ pour n'importe quel vecteur d'entiers positifs $v \in \nu(\tau)$.*

Démonstration. Pour qu'un PASS existe il est nécessaire que $rang(\tau) = s - 1$, par le théorème 1. C'est à dire que l'espace vide de τ a une dimension de un, et nous pouvons trouver un scalaire c tel que $q = cv$. De plus, si ces deux vecteurs sont des vecteurs d'entiers, alors c est rationel et nous pouvons écrire $c = \frac{n}{d}$ où $n, d \in \mathbb{N}$. Alors, $dq = nv$. Par le théorème 4, un PASS de période $p = 1^T q$ existe si et seulement si un PASS de période $dp = 1^T(dq)$ existe. Par le même théorème, un PASS de période dp existe si et seulement si un PASS de période $r = 1^T v$ existe. □

Les quatres précédents théorèmes et leurs corollaires ont une grande importance pratique. Nous avons spécifié un classe très large d'algorithmes, désignés sous la classe des algorithmes S. Étant donné un vecteur d'entiers positifs dans un espace vide de la matrice topologique, ces algorithmes trouvent un PASS avec une période égale à la somme des éléments dans q. Le théorème 3 garantit que ces algorithmes trouveront un PASS s'il existe. Les théorèmes 1 et 2 assurent qu'un tel vecteur q existe si un PASS existe. La corollaire au théorème 4 nous dit que le vecteur d'entiers positifs obtenus que nous utilisons, issus de l'espace vide la matricie topologique, nous pouvons simplifier notre système en utilisant le plus petit vecteur de ce type, obtenant ainsi un PASS avec une période minimale.

Étant donné ces théorèmes, nous pouvons donner un algorithme simple permettant de trouver un PASS appartenant à la précédente classe.

1. Résoudre pour le plus petit vecteur d'entiers positifs $q \in \nu(\tau)$.
2. Former une liste ordonnée arbitrairement L des nœuds du système.
3. Pour chaque $\alpha \in L$, ordonnancer α s'il est exécutable, en essayant chaque nœud une seule fois.

4. Si chaque nœud α a été ordonnancé q_α fois nous arrêtons.
5. Si aucun nœud dans L ne peut être ordonnancé, alors il y a un blocage.
6. Autrement aller à 3 et répéter.

Puisque le temps d'exécution est le même pour n'importe quel PASS, aucun algorithme en produira un plus rapide que cet algorithme. Par contre, il existe des algorithmes dans cette classe qui permettent de construire des ordonnancements minimisant le nombre de buffers nécessaire entre les nœuds. En utilisant la programmation dynamique ou de la programmation linéaire en nombre entier, de tels algorithmes peuvent être construits. Jusqu'à présent nous n'avons considéré qu'un ordonnancement "mono-processeur", pour une description détailler afin d'effectuer un ordonnancement "multi-processeurs" le lecteur est invité à lire [53].